UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Echtzeitsimulation von Gras mit Kollisionsbehandlung

Diplomarbeit

zur Erlangung des Grades einer Diplom-Informatikerin
im Studiengang Computervisualistik

vorgelegt von

Annabell Langs

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller Institut für Computervisualistik, AG Computergraphik
Zweitgutachter:	Dipl. Inform. Jakob Bärz Institut für Computervisualistik, AG Computergraphik

Koblenz, im September 2010

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Diplomarbeit
Annabell Langs
(Matr.-Nr. 205 210 184)

Thema: **Echtzeitsimulation von Gras mit Kollisionsbehandlung**

Realistische Naturszenen stellen einen wichtigen Aspekt heutiger Computerspiele dar. Die Darstellung von Gras in Echtzeitanwendungen kann mittels verschiedener Methoden realisiert werden – von simplen Grastexturen bis hin zu mehreren einzelnen Grashalmen, die im Wind wehen. Die meisten Computerspiele favorisieren eine mehr oder weniger statische Repräsentation von Gras, die es dem Spieler nicht erlaubt, ihre Form oder Animation zu ändern, obwohl der momentane Trend hin zu mehr Interaktivität durch physikalische Interaktionen geht. Anstatt die Grafik weiter zu verbessern, was verglichen zur tatsächlich sichtbaren Verbesserung der grafischen Qualität beim heutigen Stand der Technik ein kostspieliges Unterfangen ist, wird zunehmend eine Physiksimulation ergänzt, die von den Spielern sofort wahrgenommen wird.

Zusätzlich zu einer ansprechenden visuellen Repräsentation von Gras besteht ein weiterer wichtiger Teil dieser Arbeit in der Integration von Interaktivität: Das Gras soll auf Kollisionen mit anderen Objekten reagieren. Daher ist es notwendig verschiedene Methoden der Kollisionserkennung und -reaktion hinsichtlich ihrer Performanz in einer Echtzeitanwendung zu testen. Weiterhin sollen verschiedene Verfahren zur Beschleunigung einer glaubwürdigen Darstellung und Physiksimulation dieses Szenarios implementiert und getestet werden. Daher ist die Evaluation der Performanz des fertigen Prototyps ein weiterer Baustein dieser Arbeit, die zu einem Ausblick auf eine mögliche Anwendbarkeit in einer größeren Echtzeitanwendung führen soll.

Schwerpunkte dieser Diplomarbeit sind:

1. Recherche und Analyse vorhandener Grassimulationen und Physiksimulationen
2. Implementation einer prototypischen Echtzeitanwendung, die eine ansprechende Darstellung und Physiksimulation von Gras bietet
   - Implementation und Evaluation von Techniken zur Verbesserung der Performanz für die Anwendung auf Graslandschaften
   - Implementation und Evaluation von Strategien der Kollisionserkennung und -reaktion und Prüfung auf Eignung der Implementation auf der GPU
3. Dokumentation und Evaluation der erreichten Darstellung und Performanz der Applikation

Koblenz, den 1. April 2010

S. Müller

- Prof. Dr. Stefan Müller-

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

I Gras in der Computergrafik **1**

1 Motivation **1**

2 Visualisierung von Gras **2**
2.1 Verschiedene Renderingverfahren 3
2.1.1 Geometriebasierte Verfahren 3
2.1.2 Bildbasierte Verfahren 4
2.1.3 Volumenbasierte Verfahren 7
2.2 Level of Detail 8
2.2.1 Verfahren für Graslandschaften 9
2.3 Beleuchtung 11

3 Animation von Gras **12**
3.1 Gesamte Wiese (Wind) 13
3.1.1 Prozedurale Methoden 13
3.1.2 Physikalische Simulationen 14
3.2 Einzelne Gräser (Kollisionen) 15
3.2.1 Prozedurale Methoden 16
3.2.2 Physikalische Simulationen 16

4 Konzeptfindung und -idee **17**
4.1 Anforderungen 17
4.2 Konzeptidee 18
4.2.1 Visualisierung 18
4.2.2 Animation 19

II Theoretische Grundlagen **23**

5 Physiksimulationen **23**
5.1 Partikelsystem 23
5.1.1 Größen eines Partikels und physikalische Gesetze . . 23
5.1.2 Die Integration 24

5.1.3 Constraints 25
5.2 Starrkörpersimulation 26
5.2.1 Rotationen und verwandte Größen 26
5.2.2 Das zweite newtonsche Gesetz für Starrkörper 26
5.2.3 Kugel und Würfel 27

6 Kollisionserkennung **28**
6.1 Die grobe Kollisionserkennung 28
6.2 Die genaue Kollisionserkennung 29
6.2.1 Kollisionen zwischen Partikeln und Starrkörpern . . 30
6.2.2 Kontaktgenerierung Partikelconstraints 31
6.2.3 Sonstige Kollisionen 31

7 Kollisionsbehandlung **33**
7.1 Geschwindigkeitsänderung 33
7.1.1 Das Kontakt-Koordinatensystem 34
7.1.2 Die gewünschte Geschwindigkeitsänderung 35
7.1.3 Anwendung des Impulses 36
7.2 Positionsänderung 36
7.2.1 Lineare Projektion 37
7.2.2 Nichtlineare Projektion 37

III Praktische Umsetzung **39**

8 Visualisierung **40**
8.1 Geometrie rendern 41
8.2 Texturierung 42
8.3 Level-of-Detail-System 43
8.3.1 Kamerabewegung 44
8.4 Gras 46
8.4.1 Generierung 46
8.4.2 Aktualisierung der Daten bei Kamerabewegung . . . 48
8.5 Kantengras 49
8.5.1 Generierung 51

8.5.2 Aktualisierung der Daten bei Kamerabewegung . . . 51
8.6 Bodentextur und Terrain 54
8.7 Bildqualitätsverbesserungen 54
8.7.1 Screen Space Ambient Occlusion 54
8.7.2 Alpha To Coverage 56

9 Animation **57**
9.1 Initialisierung des Physiksystems 58
9.2 Ablauf pro Frame 60
9.2.1 Windkräfte 60
9.2.2 Kollisionserkennung 61
9.2.3 Kollisionsbehandlung 62
9.2.4 Partikel- und Starrkörperintegration 63
9.2.5 Partikelconstraint 63
9.2.6 Abschließende Aktualisierung des Grases 64
9.3 Interpolation 65
9.4 Berechnungen auf der GPU 67

10 Grafische Benutzeroberfläche **69**
10.1 Wiese und Terrain ändern 69
10.2 Interaktive Parametereinstellungen 69

11 Zusätzliche Features **71**
11.1 Konfigurationsdatei 71
11.2 Fußballfeld 72
11.3 Rasenmäher 72

IV Ergebnisse und Bewertungen **75**

12 Darstellungs- und Animationsqualität **75**
12.1 Vergleich zu bisherigen Verfahren 75
12.1.1 Darstellungsqualität 76
12.1.2 Animationsqualität 77
12.2 Vergleich mit der Realität 80

13 Performance **81**
13.1 GPU-Berechnung 82
13.2 Interpolation 82
13.3 Grobe Kollisionserkennung 84
13.4 Level-of-Detail-System 84
13.5 Gesamtperformance 84

14 Vor- und Nachteile der Grassimulation **85**

15 Verbesserungsmöglichkeiten **88**

16 Zusammenfassung und Ausblick **92**

Abbildungsverzeichnis **95**

Literatur **99**

Teil I
Gras in der Computergrafik

1 Motivation

Schon lange versucht die Computergrafik die Realität möglichst genau abzubilden. Die Nachbildung der Natur stellt dabei eines der größten Probleme dar. Nicht nur ihre Darstellung ist essentiell um in virtuelle Welten vollständig eintauchen zu können. Auch ihre Simulation spielt eine immer wichtigere Rolle im Erlebnisgefühl. Virtuelle Welten können dabei tagelang gerenderte Animationsfilme wie z. B. *Shrek* oder *Big Buck Bunny* sein, Filme mit teilweise computeranimierten Szenen wie z. B. *Avatar - Aufbruch nach Pandora* oder Computerspiele wie z. B. *Crysis* oder *Anno 1404*. In den genannten Beispielen liegt ein Schwerpunkt auf der Darstellung einer besonders glaubhaften, realistischen und schönen Abbildung und Simulation von Natur aus der Wirklichkeit oder Fiktion.

Bei Naturszenen spielt dabei oft Gras eine entscheidene Rolle um der virtuellen Welt Leben einzuhauchen. Gras kommt meist in schier unendlichen Mengen in Form von Wiesen vor. Daher ist es insbesondere fordernd, sehr viele Gräser darzustellen und zu simulieren. Auch die scheinbar chaotische Anordnung von einzelnen Grashalmen, die im Gesamtbild erst den Eindruck von Gras entstehen lässt, kann als Schwierigkeit für die Computerberechnung angesehen werden. Daraus ergibt sich die Eigenschaft der Anisotropie[1] von Gras [Bak03], die in Zusammenhang mit der Beleuchtung das Aussehen stark beeinflusst. Einzelne Grashalme oder gar die gesamte Wiese müssen gleichzeitig aber auch auf ihre Umwelt reagieren. Dies kann einerseits durch Berührung, also Kollision, geschehen, die verschieden große Areale der Wiese betreffen, andererseits auch durch Kräfte, die nicht über Kollisionen wirken, wie dem Wind, der dann die gesamte Wiese beeinflusst.

In Filmen, in denen für die Darstellung und Simulation von Gras genug Rechengeschwindigkeit durch Servercluster zur Verfügung steht, ist die Entwicklung in den letzten Jahren weit fortgeschritten und eine Unterscheidung zwischen Realtität und Virtualität kaum mehr auszumachen. In Echtzeitsimulationen wie beispielsweise Computerspielen hingegen stellt sie immer noch ein großes Problem dar. Johan Andersson sieht in seinem Vortrag *5 Major Challenges in Interactive Rendering* auf der *ACM Siggraph 2010* die prozedurale Verteilung von Blattwerk und Laub mit Verweis auf ein Bild, das eine Graslandschaft zeigt, als „the single most important [topic] for game development“ (siehe Folien S. 28, [And10]) an. Dass sich aber auch in Computerspielen die Darstellungsqualität in den letzten Jahren verbessert hat, ist nicht von der Hand zu weisen. Waren es zunächst flache

[1] Richtungsabhängigkeit

Abbildung 1: Gras im Animationsfilm *Big Buck Bunny*

Texturen, die versuchten die Struktur von Gras grob abzubilden, so kamen später einzelne Grasbüschel hinzu. Aktuell werden Wiesen immer noch als Ansammlungen von nunmehr sehr vielen Grasbüscheln dargestellt.

Neben der Darstellungsqualität, die sich also merklich verbessert hat, wird die Simulation von Gras nach wie vor eher stiefmütterlich behandelt. Meistens beschränkt sich diese auf die Bewegung von Gras auf Grund von Wind. Eine genauere Simulation z. B. auf Grund von Kollisionen findet zumeist nicht statt.

Da also die Darstellung und insbesondere die Simulation von Gras in Echtzeitanwendungen ein für neue Lösungen noch recht offenes Forschungsgebiet ist, ergibt sich ein für diese Diplomarbeit breites Aufgabenspektrum.

Im weiteren Teil I wird zunächst auf bisherige Visualisierungsverfahren (Kapitel 2) und Animationsverfahren (Kapitel 3) eingegangen. Danach folgt die Konzeption einer eigenen Lösungsstrategie des Problems (Kapitel 4) und damit einhergehende theoretische Grundlagen (Teil II). Im Anschluss wird die eigentliche Implementation (Teil III) vorgestellt und zum Schluss das erzielte Ergebnis bewertet (Teil IV).

2 Visualisierung von Gras

Viele der folgenden Verfahren beschäftigen sich explizit mit der Darstellung von Gras. Eng verwandt mit dem Problem der Grasvisualisierung ist aber auch die Darstellung von Fellen oder Haaren. Bereits 1989 widmeten sich Kajiya und Kay dem Rendern von Fell [KK89]. Von Neyret folgte 1998 eine Veröffentlichung zum Thema *Modeling, Animating and Rendering Com-*

plex Scenes using Volumetric Textures [Ney98], in der als Beispiel ein Rasen unter Einwirkung von Wind zu sehen ist. Diese stellt eine der ersten Arbeiten dar, die explizit die Darstellung von Gras zeigt. Deutlich wird, dass sich auf Grund der langen Zeitspanne bis heute schon etliche Forscher mit diesen Themen beschäftigt haben und dadurch ganz unterschiedliche Verfahren enstanden sind. Im Folgenden kann daher nur auszugsweise auf die wichtigsten und interessantesten Lösungen samt ihrer Vor- und Nachteile eingegangen werden.

2.1 Verschiedene Renderingverfahren

Alle vorgestellten Verfahren zur Darstellung von Gras (oder auch Fell bzw. Haaren) lassen sich grob in drei Kategorien [Bou08] einordnen:

- Geometriebasierte Verfahren
- Bildbasierte Verfahren
- Volumenbasierte Verfahren

Die bekanntesten Vertreter werden in den nächsten Unterkapiteln vorgestellt.

2.1.1 Geometriebasierte Verfahren

Unter geometriebasierten Verfahren versteht man die Darstellung von Gras durch aufwändige Geometrie. Dadurch ergibt sich eine große Anzahl an Polygonen und damit eine sehr genaue Wiedergabe der Wirklichkeit. Durch die Nutzung vieler Dreiecke für einen einzelnen Grashalm ist eine präzise und vielfältige Darstellungsweise möglich. Da die gesamte Geometrie vorhanden ist, kann das Gras zudem genau beleuchtet werden [Bou08]. Da allerdings eine Wiese aus Millionen von Grashalmen besteht, kann dieser Ansatz in Echtzeitsimulationen nur für einen begrenzten Bereich, z. B. in Nähe des Betrachters, verwendet werden (daher oftmals in Verbindung mit sogenannten Level-of-Detail-Verfahren, siehe Kapitel 8.3). In der Filmindustrie, in der das Rendern eines Frames[2] eines vollständig computeranimierten Films mehrere Stunden benötigen kann, ist die Verwendung von viel Geometrie hingegen kein Problem. Große Wiesen sind mit geometriebasierten Verfahren also nur im Offline-Rendering-Bereich realisierbar.

Das Erzeugen von Geometrie in dieser Menge wäre von Hand eine zeitlich sehr aufwändige Arbeit und kaum zu bewältigen. Daher wird hierfür auf prozedurale Verfahren zurückgegriffen. Nach [Bou08] ermöglichen Partikelsysteme (siehe auch Kapitel 5.1) und Hermitkurven das Generieren der Koordinaten der Geometrie, deren Parameter wiederum individuell

[2] Einzelbild

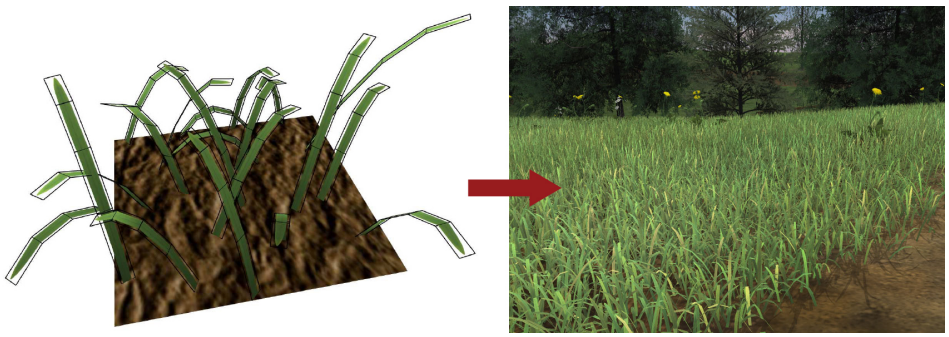

Abbildung 2: Geometriebasiertes Verfahren (für die feinste Detailstufe, nach [Bou08])

eingestellt werden können. Banisch [Ban06] führt exponentielle Funktionen als weitere prozedurale Möglichkeit auf.

Perbet [PC01], Colditz [CCDH05] und [Bou08] verwenden in ihren Arbeiten zumindest für einen Teil ihrer Darstellung echte Geometrie. Diesen Konzepten liegt ein Level-of-Detail-Verfahren zu Grunde, das heißt, dass das Geometrieverfahren nur in dem Betrachter am nächsten liegenden Detaillevel verwendet wird. Perbet verwendet für die 3D-Geometrie aneinandergereihte Streckenprimitive. Colditz, der sich auf Bäume bzw. Wälder (und damit ein ähnliches Problem wie bei Gras) in Naturszenen konzentriert hat, verwendet Modelle mit voller Geometriekomplexität ausschließlich für die Nähe (S. 6, [CCDH05]). Auch Boulanger nutzt für einzelne Grashalme im Nahbereich geometriebasierte Modelle, genauer gesagt pro Halm einen gebogenen viereckigen Streifen mit teilweise transparenten Texturen (siehe Abbildung 2).

2.1.2 Bildbasierte Verfahren

Anders als die geometriebasierten Verfahren, die schnell zu einer langen Renderzeit führen, ermöglichen bildbasierte Verfahren eine Entkopplung der Renderzeit von der Geometriekomplexität der Szene [SKP05]. Bildbasierte Verfahren stellen Gras mittels Texturen dar. Texturen werden dabei vielfältig verwendet. Für ferne Landschaften können sie als vereinfachte Repräsentationen verwendet werden. Zusätzliche Geometrie wird dabei nicht benötigt. Allerdings ist damit einhergehend keine Animation möglich. Da Texturen in den Anfängen der Grasdarstellung meistens die einzige Möglichkeit waren, überhaupt Gras darzustellen, findet man diese Art der Darstellung in vielen älteren Computerspielen wieder. Aber auch in den aktuellsten Titeln wird dieses Verfahren verwendet, um weit entfernte Landschaften mit Gras zu versehen. Weiterhin kommt die schlichte Grastexturierung oft in den Level-of-Detail-Systemen als gröbste Detailstufe zum Einsatz (siehe Abbildung 3). Die Texturierung von Oberflächen ist heutzutage oftmals nur ein Hilfsmittel

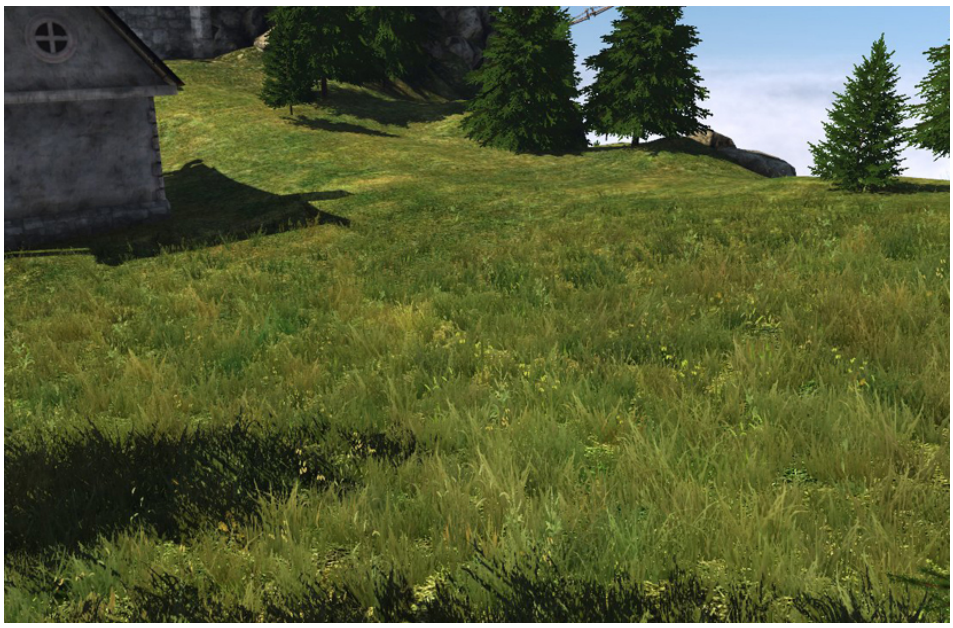

Abbildung 3: Level-of-Detail-System mit Billboards und Bodengrastextur aus *Unigine Engine Benchmark* 2010

um das tatsächliche Gras zu betonen. Auf die Oberflächentextur werden zusätzliche Rechtecke, ebenfalls mit einer (teilweise transparenten) Textur versehen, aufgestellt.

Dabei gibt es wiederum zwei Möglichkeiten diese zusätzlichen sogenannten *Texturquads* zu platzieren: statisch oder dynamisch je nach Blickwinkel (sogenannte *Billboards*).

Bei der statischen Variante werden die Vertices des Quads einmal berechnet und das mittels Textur dargestellte Grasbüschel platziert. Bei Computerspielen, in denen der Blickwinkel statisch ist (2D Spiele mit Draufsicht), ist dies eine gute Möglichkeit weitere Details zu ergänzen. Mehrere verschachtelte Texturquads (manchmal trotz ihrer statischen Positionierung bereits als Billboard bezeichnet) erhöhen die Komplexität zusätzlicher Grasbüschel (siehe Abbildung 4). Bei Computerspielen, die allerdings die freie Bewegung in der virtuellen Welt erlauben und damit die unterschiedlichsten Blickwinkel auf die Szene zulassen, könnte die Darstellung mittels Billboardeffekt hilfreicher sein um die Illusion aufrecht zu erhalten, es handele sich um ein Objekt mit Tiefe, wenngleich es sich nur um ein plattes Rechteck handelt.

Billboards sind Primitive (Rechtecke), auf die eine semi-transparente Textur gelegt wird und die immer zum Betrachter gedreht (also dynamisch) gerendert werden. Durch die Nutzung von Transparenz ist es damit möglich, Objekte auf diesen Primitiven darzustellen, die nicht die gleiche äußere Form wie das Primitiv haben, sondern beliebig geformt sein können.

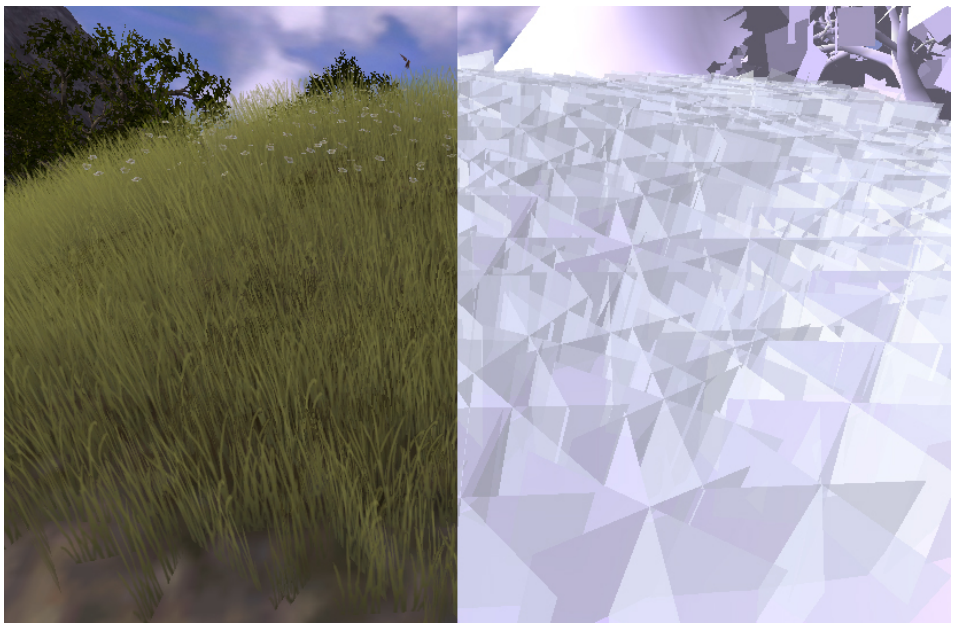

Abbildung 4: Geschachtelte Texturquads aus *Codecreatures Demo* [Pel07]

Schnell ist ersichtlich, dass Grashalme oder -büschel auf solch einer teilweise transparenten Textur gut darstellbar sind. Eine erweiterte Form von Billboards, die *Volumetric Billboards* [DN09], erlauben eine noch überzeugendere Darstellung komplexer Strukturen. Durch die Nutzung von 3D-Zellen mit volumetrischen Bildern (3D-Texturen) werden visuelle Artefakte (z. B. Parallaxeffekte[3]) normaler Billboards verhindert. Alle Billboard-Verfahren besitzen das Potenzial, durch wenige Vertices die Illusion eines aufwändigen Körpers herzustellen.

Viele Arbeiten nutzen Billboards entweder exklusiv oder in Zusammenhang mit anderen Verfahren. Letztere werden in Kapitel 8.3 betrachtet.

Pelzer stellt in seiner Veröffentlichung *Rendering Countless Blades of Waving Grass* [Pel07] eine Benchmark Applikation vor, die exzessiv von transparenten Grastexturen auf Quads Gebrauch macht (siehe Abbildung 4). Auf einer Grastextur befinden sich mehrere Grashalme in unterschiedlichen Farben und Größen. Die Grasquads werden auf Grund einer freien Kameraführung dabei nicht linear in Reihen angeordnet, sondern ineinander gekreuzt aufgestellt. Diese werden dann in der Szene dicht beieinander gestellt, zur Laufzeit von hinten nach vorne sortiert, und mit aktiviertem Alphablending[4] und Tiefentest gezeichnet.

[3] Scheinbare Veränderung der Position eines Objekts bei Bewegung des Betrachters [Wik10b]

[4] Überlagerung mehrerer Bildinformationen unter Berücksichtigung der Transparenzwerte

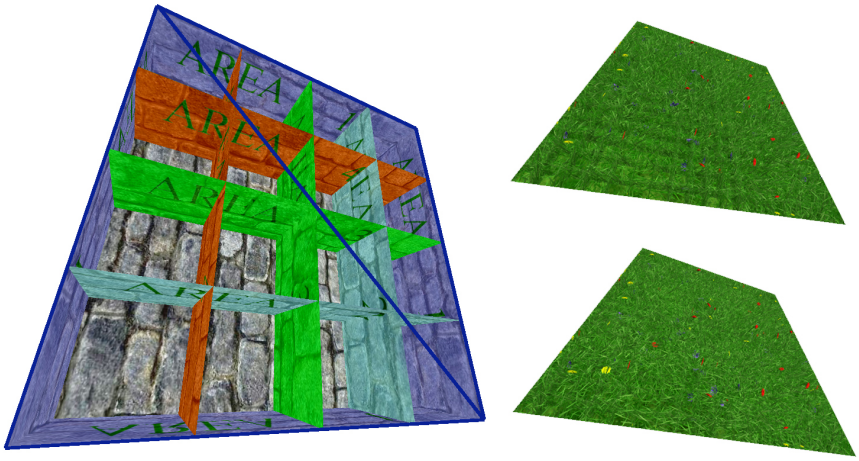

Abbildung 5: Virtuelle Bilboards: Struktur ähnlich zu Relief Mapping (links) und Resultate (rechts) mit Hervorhebung der Struktur (oben) [HWJ07]

In Orthmanns Konzept [ORSK09] werden Billboards mittels eines Geometry-Shaders[5] vorab generiert und im Grafikkartenspeicher gehalten. Auch hier werden ganze Grasbüschel auf jeweils einem semi-transparenten Texturquad abgebildet. Zusätzlich definieren drei zuvor erstellte Texturen die Anordnung, Richtung und Zufälligkeit des auf dem Terrain verteilten Grases.

Habel geht in seiner Arbeit [HWJ07] noch weiter und verwendet sogenannte *virtuelle Billboards* (siehe Abbildung 5). Diese werden auf einem *Trägerpolygon* implizit definiert und der für Alphablending benötigte Transparenzwert innerhalb der Billboards mittels Raytracing ähnlich des Relief Mappings[6] berechnet. Auf Grund des Raytracingansatzes ist ein Sortieren der Szene von hinten nach vorne nicht nötig.

2.1.3 Volumenbasierte Verfahren

Volumenrendering wird nicht nur in der Medizin verwendet, sondern auch als Ansatz für das Gras- oder Fellrendering. Die Volumen bestehen hierbei aus mehreren Ebenen, die meist parallel zum Boden übereinander angeordnet liegen. Die Implementation eines solchen volumenbasierten Rasens ist damit einfach und erzeugt einen guten Parallaxeffekt [Bou08]. Aus flachen Winkeln heraus sind die einzelnen Ebenen jedoch erkennbar und lassen sich nur durch zusätzliche Ebenen, die parallel zur Kameraebene verlaufen, weniger sichtbar machen.

Die Grundlage für die folgenden Ansätze bildete Kajiya bereits 1989

[5] Shader, der zusätzliche Geometrie erzeugen oder löschen kann

[6] Technik zum Erhöhen des Detailgrads mittels einer Textur ohne die Geometrie der Oberfläche zu ändern

[KK89] mit dreidimensionalen Texturen, die er *Texel* (nicht zu verwechseln mit dem Texel[7], daher von manchen späteren Publikationen auch Texcell genannt, siehe Abbildung 8) nannte um damit Fell zu visualisieren. Hiermit konnte bereits zur damaligen Zeit feinste Geometrie dargestellt werden.

Bakay stellte später seinen volumenbasierten Ansatz [BLH02] speziell für Gras als einer der ersten vor. Er bezeichnete die verschiedenen Ebenen als *Shells*, die entlang der Normalen des Geländes extrudiert werden. Ihm reicht dabei eine Basistextur, die kopiert wird und ihre Vertices mittels eines Vertex-Shaders versetzt werden. Jede Textur einer Ebene ist nur an jenen Stellen mit Farbe gefüllt, wo die Grashalme sichtbar sein sollen, und an allen anderen Positionen transparent. Dadurch ist es ohne Mehraufwand möglich beliebig viele einzelne Grashalme darzustellen. Diese Technik ist laut eigener Aussage für Perspektiven von oben wie beispielsweise in einem Flugsimulator oder für kurzes Gras geeignet. Decaudin [DN04] erweitert die Idee der Schichten um *Silhouettentexcells*, die wie oben genannt das Problem der Sichtbarkeit der einzelnen Ebenen bei flachen Blickwinkel verhindern. Da er Wälder auf Bergen darstellen musste, war eine Lösung für dieses Problem unabdingbar.

Nicht nur zum Boden parallel liegende Ebenen werden unter dem Begriff volumenbasierte Verfahren verstanden. Auch zur Kameraebene parallel liegende (axis-aligned) Texturen ähnlich der bildbasierten Verfahren können hintereinander aufgestellt werden und für weiter entfernte Bereiche den Eindruck von Tiefe vermitteln. Boulanger verwendet solche Texturstaffelungen in seiner Grasimplementation [Bou08]. Manchmal werden so genutzte Texturen auch als *volumetrische Texturen* bezeichnet.

2.2 Level of Detail

Immer dann, wenn der Detailreichtum einer Szene besonders groß ist und damit zu viel potenzielle Geometrie gezeichnet werden muss, ist eine Reduktion dieser entscheidend um eine flüssige Bilddarstellung bzw. Bildwiederholrate garantieren zu können. Insbesondere bei großen virtuellen Außenarealen, bei denen keine Wände oder ähnliches die Weitsicht versperren, ist dies notwendig. Weit entfernte Objekte entsprechen dann nur noch wenigen Pixeln[8] auf dem Monitor. Entweder wird ein solches Objekt dann komplett ausgeblendet oder in einer veränderten, reduzierten Repräsentation dargestellt. Anwendungsbeispiele für Level-of-Detail-Verfahren finden sich also vor allem im Bereich von Naturlandschaften wieder. Eine solche Anwendung ist die Generierung und Darstellung von Terrain. In diesem Forschungsgebiet wurden schon viele unterschiedliche Verfahren entwickelt um weit entferntes Terrain mit einer anderen Detailstufe (Level)

[7] In Anlehnung an Pixel ein Punkt auf einer Textur

[8] Kunstwort aus den Wörtern *picture* (bzw. umgangssprachlich *pix*) und *element*, deutsch Bildpunkt

darzustellen als nahes. Bekannte Verfahren sind z. B. *Real-time Optimally Adapting Meshes* [Whi08], *Chunked Level of Detail* [Tha02] oder *Geometric Mipmaps* [dB00]. Sie ermöglichen die Beschleunigung der Darstellung und Verringerung des Speicherbedarfs für die Geometrie.

Neben den genannten Vorteilen entstehen durch Level-of-Detail-Systeme auch neue Probleme. Eine Schwierigkeit bei Level-of-Detail-Verfahren entsteht bei den Übergängen zwischen verschiedenen Detailstufen. Diese dürfen im besten Fall nicht auffallen, d. h. sie müssen ineinander übergehen, da das menschliche Auge solche plötzlichen Änderungen in der Szene sofort wahrnimmt. Weiterhin ist die Berechnung dieser verschiedenen Detailstufen anhand der Entfernung zur Betrachterposition zur Laufzeit auch ein Performanceproblem. Ohne Level-of-Detail-Systeme kann die gesamte Szenengeometrie im Voraus berechnet werden. Nun müssen Teile der Szene während der laufenden Simulation hinzugefügt oder entfernt werden.

Neben dem Einsatzgebiet für Terrains haben sich Level of Detail Systeme auch für Graslandschaften etabliert. Einige wichtige Vertreter werden im Folgenden vorgestellt.

2.2.1 Verfahren für Graslandschaften

Der älteste Vertreter eines Level-of-Detail-Systems für Gras stammt von Perbet [PC01]. Er verwendet drei Detailstufen. In der ersten Stufe, 3D genannt, werden aneinandergereihte Streckenprimitive verwendet. Für die mittlere Entfernung, bei Perbet als 2,5D bezeichnet, verwendet er volumetrische Texturen. Für die am entferntest liegende Stufe (2D) verwendet er eine einfache Bodentextur. Diese Aufteilung, von geometriebasierten Verfahren in der Nähe hin zu bildbasierten Verfahren in der Ferne, wird auch in späteren Veröffentlichungen oft verwendet. Die gesamte Wiese wird in Kacheln von Graspatches aufgeteilt. Das Maß, wann welcher Detaillevel verwendet wird, ist dann die diskrete Distanz zwischen dem darzustellenden Gras und dem Betrachter. Gras, das sich eine Kante oder ein Vertex mit einem anderen Grasstück teilt, hat die Distanz 1. Die 2,5D-Repräsentation wird dann beispielsweise für die Distanz 3 bis 5 verwendet. Die Kachelung erlaubt es zusätzlich einfaches 2D-Culling[9] anzuwenden um weiteres Gras nicht zu zeichnen, welches nicht im Sichtfeld liegt. Bei den Levelübergängen muss beachtet werden, dass immer das gleiche Graspatch in unterschiedlichen Repräsentationen gerendert wird. Daher werden die verschiedenen Darstellungsformen von einander abhängig generiert: zunächst wird die 3D-Variante erstellt, danach von dieser Repräsentation ausgehend die Textur für die 2,5D-Darstellung berechnet und am Ende eine solche unter Berücksichtigung der 2,5D-Textur für die letzte 2D-Stufe generiert.

Ähnlich zu Perbet verfolgt auch Colditz [CCDH05] den Level-of-Detail-

[9] Verwerfen von nicht Sichtbarem zur Leistungssteigerung

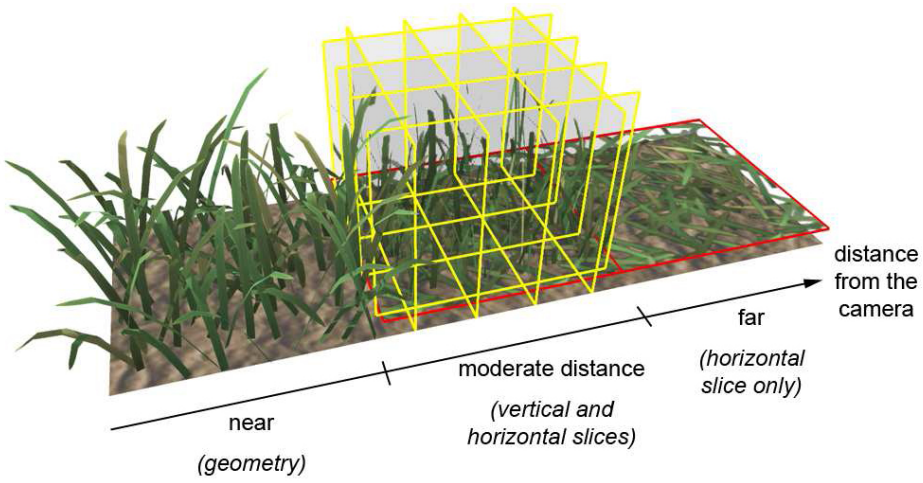

Abbildung 6: Level-of-Detail-System, welches in [Bou08] zum Einsatz kommt

Ansatz mit geometriebasiertem Rendering für die Nähe und bildbasiertem Verfahren (Billboards) für die Ferne. Behrendt [BCF+05] nutzt für seine Landschaftsdarstellung zusätzlich zu Billboards noch das volumetrische Shellverfahren. Dabei teilt er beide Verfahren noch einmal in feinere und gröbere Repräsentationen auf. So verwendet er einfachere Kreuzbillboards, die nur aus zwei Texturquads bestehen, und komplexe mehrteilige Billboards. Zusätzlich benutzt er bei der gröbsten Detailstufe unterschiedlich viele Ebenen für den volumenbasierten Shellansatz, denn der Abstand zwischen den einzelnen Ebenen ist bei großem Abstand zum Betrachter im Screen Space[10] sehr klein und kann daher abhängig von der Distanz unterschiedlich gewählt werden.

Ein besonders realistisches Ergebnis erzielt Boulanger in seiner Dissertation [Bou08]. Trotz verschiedenster Blickwinkel ist das Erkennen eines Level-of-Detail-Systems nicht möglich. Um dies zu erreichen, verwendet er ebenfalls drei Detailstufen: geometriebasierte texturierte viereckige Streifen als Grashalme, semi-transparente ineinander verkreuzte vorberechnete Texturschichten (*vertical slices*) und eine einfache Textur für weit entfernte Grasareale (*horizontal slices*). Entscheidend ist nun ein weicher Übergang zwischen diesen Detaillevels. Hierzu gewichtet er die einzelnen Verfahren anhand der Distanz zur Kamera. In Übergangsbereichen verwendet er anteilig beide benachbarten Detailstufen, bis der eine Level komplett in den nächsten übergeht.

[10] Bildraum, also der auf dem Monitor dargestellte Bereich

Abbildung 7: Fotografiertes Gras mit erkennbaren Glanzpunkten und Verschattungen

2.3 Beleuchtung

Die Beleuchtung von Gras ist wichtig und schwierig, da Gras genau wie Fell oder Haare anisotrope Eigenschaften hat [Bak03]. Durch die Sonneneinstrahlung und den vielfältigsten Ausrichtungen der einzelnen Grashalme entstehen an verschiedenen Stellen des Grases beispielsweise Glanzpunkte oder Verschattungen (vergleiche Abbildung 7). Schon von jeher ist die Visualisierung von anisotropen Materialien oder Strukturen im Vergleich zu isotropen besonders aufwändig. Da gerade die Beleuchtung bei anisotropen Oberflächen einen entscheidenden Einfluss auf das Aussehen hat, ist auch die Beleuchtung von Gras ein wichtiger Bestandteil jeder Visualisierung.

Je nach verwendeter Darstellungsweise können sich die eingesetzten Beleuchtungsverfahren stark voneinander unterscheiden.

Nach Shah [SKP05] approximieren die meisten geometriebasierten Ansätze die Beleuchtung durch einen einfachen diffusen und spekularen Reflexionsterm. Die komplexeren Lichteffekte wie z. B. Sub Surface Scattering oder Interreflexionen und Selbstverschattungen können damit nicht abgebildet werden – sie wären bei den geometriebasierten Verfahren für Echtzeitsimulationen zu rechenaufwändig.

Bildbasierte Verfahren hingegen können ohne größeren Aufwand solche Effekte darstellen. Behrendt [BCF+05] benutzt hierzu für seine Billboards Spherical Harmonics Basisfunktionen[11] um die Reflexionsfunktion zu approximieren. Damit können komplexe bidirektionale Reflexionsverteilungs-

[11] Eine Menge an einfach zu beschreibenden Funktionen, die als Linearkombination eine meistens schwierigere Funktion approximiert.

funktionen (BRDF), beliebig viele Lichtquellen und indirekte Beleuchtung verwendet und auf der GPU berechnet werden. Schatten werden zusätzlich mittels *Shadow Mapping* berechnet. Eine ähnlich aufwändige Beleuchtung ist auch mit sogenannten *Bidirectional Texture Functions* möglich, wie sie von Boulanger [Bou08] verwendet werden. Für eine Ebene seiner Volumenrepräsentation wird das Gras aus fünf verschiedenen Richtungen beleuchtet und aus zwei Blickrichtungen (von vorne und von hinten) gesehen in einer Textur gespeichert. Damit ist eine Beleuchtung pro Pixel, teilweise mit zusätzlicher Interpolation zwischen Licht- und Blickrichtungen, möglich.

Eine einfachere Möglichkeit der Beleuchtung von Billboards besteht darin, dass die verwendeten Texturen direkt die Beleuchtungsinformationen speichern. Das heißt, dass die Grastextur direkt beleuchtet abgespeichert wird. Sie beinhaltet dann allerdings nur eine statische Beleuchtung. Dieses Verfahren wird oft bei einfachen Bodentexturen angewendet. Schatten lassen sich damit ähnlich leicht simulieren, indem in den unteren Bereichen der Textur dunklere Farbtöne verwendet werden [Bou08]. Eine dynamische Beleuchtung ist nur dann möglich, wenn anstatt statisch vorberechneter fertiger Texturen die für die Lichtberechnung benötigten Normalen zusätzlich in einer Textur gespeichert werden und die Beleuchtung der Billboards dann anhand der Lichtquellen zur Laufzeit berechnet wird.

Um dynamische globale Beleuchtung zu erzielen nutzt Orthmann [ORSK09] vorberechnete Volumen, in denen pro Voxel ambiente Verdeckungsinformationen und Bestrahlungsstärkeinformationen für die jeweilige Position in der Szene gespeichert sind. Zusammen mit den Billboardtexturen wird auf der GPU der finale Farbwert berechnet.

3 Animation von Gras

Ohne Animation würde Gras sehr unrealistisch wirken. Da Gras vor allem in der Natur zu finden ist, ist es dort allen Umwelteinflüssen ausgesetzt. Gras bewegt sich, wenn Wind vorhanden ist oder wenn eine andere Kraft es zu Boden drückt. Als solch eine Kraft wird hier ein Objekt angesehen, welches Kontakt zur Wiese hat.

Wind, also ein Phänomen, das die gesamte Wiese in ihrer Bewegung beeinflusst, wurde dabei schon oft untersucht und in Grassimulationen implementiert. Einzelne Kollisionen hingegen fanden bisher weniger Beachtung, obgleich sie ebenso entscheidend für einen natürlichen Eindruck sind. Insbesondere in Computerspielen fällt auf, dass auf eine solche Kollisionsbehandlung zwischen Gras und sonstigen Objekten verzichtet wird. Dies ist vor allem dann zu bemerken, wenn virtuelle Spielfiguren in Naturlandschaften durch hohes Gras laufen und dieses nur durch Wind, nicht aber durch den Avatar, bewegt wird. Erste Ansätze der Kollisionsbehandlung mit Vegetation sind in *Crysis* zu sehen: bestimmte Pflanzenarten reagieren

bei Berührung mit dem Spieler. Eine Kollision mit Gras findet aber auch hier nicht statt.

Daher wird die weitere Animation des Grases in zwei Kategorien eingeteilt und getrennt betrachtet. Manche Implementationen lösen beide Probleme mit ein und dem selben Verfahren. Diejenigen, die insbesondere die Kollisionen mit Objekten behandeln, werden im letzten Unterkapitel 3.2 behandelt.

3.1 Gesamte Wiese (Wind)

Zur Animation der gesamten Wiese können grundsätzlich prozedurale oder physikbasierte Methoden zum Einsatz kommen. Unter prozeduralen Methoden versteht man solche, die zur Laufzeit die Bewegungen des Grases mittels Algorithmen berechnen [Wik10c]. Sie werden vor allem bei Echtzeitanwendungen verwendet (S. 36, [Bou08]). Animationsverfahren, die auf Physiksimulationen setzen, sind ebenfalls möglich. Dabei wird das Gras oft approximiert und auf größerem Maßstab animiert anstatt jeden einzelnen Grashalm zu simulieren. Die Anwendbarkeit der Verfahren hängt wiederum von der Darstellungsweise des Grases ab.

Nachfolgend werden einige Ansätze exemplarisch vorgestellt.

3.1.1 Prozedurale Methoden

Auf Grund der einfachen Struktur von Billboards werden zur Animation dieser oft prozedurale Verfahren verwendet. Die einfachste Möglichkeit Bewegung in eine mit Billboards bzw. Texturquads erstellte Wiese zu bekommen ist die Positionsänderung dieser mittels trigonometrischer Funktionen, beispielsweise der Sinus- oder Kosinusfunktion. Um dies zu leisten verwendet Pelzer [Pel07] die Funktionalität von Vertex-Shadern aktueller Grafikkarten, um die oberen Vertices eines Quads zu ändern. Er unterscheidet zwischen der uniformen Animation von mehreren Grasobjekten, der Animation einzelner Vertices eines Grasquads und der individuellen Animation pro Grasobjekt. Letzteres Vorgehen stellt für Pelzer die „goldene Mitte“ dar, wenngleich alle Methoden ihre Vor- und Nachteile besitzen.

Auch Habels virtuelle Billboards [HWJ07] können mittels versetzter Texturzugriffe bewegt werden. Dies ist allerdings auf Grund der besonderen Verwendung der Billboards eine Sonderlösung und auf die übliche Billboardnutzung nicht übertragbar.

Perbet verwendet für sein geometrie- und bildbasiertes Level-of-Detail-System [PC01] sogenannte Windprimitive, die als Sender verstanden werden können und Informationen an Empfänger vermitteln. Beide Grasrepräsentationen, Geometrie und Texturquads, erhalten einen Empfänger, der mit den 3D-Grashalmen bzw. den vertikalen Kanten des Texturquads verknüpft ist. Die Informationen beinhalten die Richtung, in die sich das

Gras knicken soll, und wie stark sich das Gras biegen soll. Es werden die resultierenden Effekte modelliert anstatt Wind als physikalische Kraft zu verstehen. In jedem Frame wird dann für jedes zu animierende Grasobjekt sein entsprechender Empfänger ausgewertet, der evtl. auch von mehreren Sendern Parameter erhalten hat.

Auch für volumenbasierte Grasdarstellungen gibt es Möglichkeiten, das Gras zu animieren. Bereits 1998 hat sich Neyret [Ney98] mit der Animation der Texcells beschäftigt. Das Volumen wird dabei in Patches, die Texcells, unterteilt und die Höhenvektoren entlang der vertikalen Patchkanten können dann deformiert werden. Eine Jitteringfunktion[12] im Sinne eines Kraftfeldes kann verwendet werden um eine natürlichere und weniger reguläre Verteilung der Höhenvektoren zu erreichen. Solche Kraftfelder können mittels Physiksimulationen oder empirischer Funktionen erstellt werden und dem Wind nachempfunden werden. Da alle Texcells zusammenhängen, muss auf Auswirkungen auf benachbarte Texcells ein besonderes Augenmerk gelegt werden. Boulanger spricht daher von einer nötigen Synchronisation zwischen den Texcells und einer damit einhergehenden uniformen Bewegung großer Bereiche der volumetrischen Texturen [Bou08].

Auch Bakay [BLH02] hat ein Verfahren entwickelt um sein Volumengras zu bewegen. Dabei wird jeder Vertex der unterteilten Ebenen entsprechend ihrer Normalen und eines Windvektors, der senkrecht zum Normalenvektor steht und für jedes Vertex abgespeichert wird, bewegt. Vertices zwischen den Schichten behalten ihre jeweiligen vertikalen Abstände zueinander und damit bleibt auch die Länge der Grashalme konstant.

3.1.2 Physikalische Simulationen

Physiksimulationen wurden bisher selten zur Animation von Gras herangezogen, da ihre Berechnung zu aufwändig erscheint. Tatsächlich ist die genaue physikalische Berechnung aller Grashalme einer Wiese auch für heutige Prozessoren und Grafikkarten zu arbeitsintensiv, wie eine eigene Untersuchung im Laufe der Konzeptfindung ergeben hat (siehe Kapitel 4). Dennoch können vereinfachte physikalisch basierte Modelle genutzt werden um ein approximiertes Ergebnis für die Bewegung zu erhalten.

Banisch [Ban06] verwendet hierzu ein Feder-Masse-System. Feder-Masse-Systeme lösen gewöhnliche Differentialgleichungen (engl. ODE für ordinary differential equations) auf Basis des Newtonschen Grundgesetztes $\vec{F} = m\vec{a}$. Das Grasfeld wird dabei als eine Menge von Massepunkten und Federn in einer festen Topologie modelliert. Mehrere mögliche Topologien (siehe auch Abbildung 8) wurden entwickelt:

[12] Funktion zum Hinzufügen von Schwankungen/Unregelmäßigkeiten

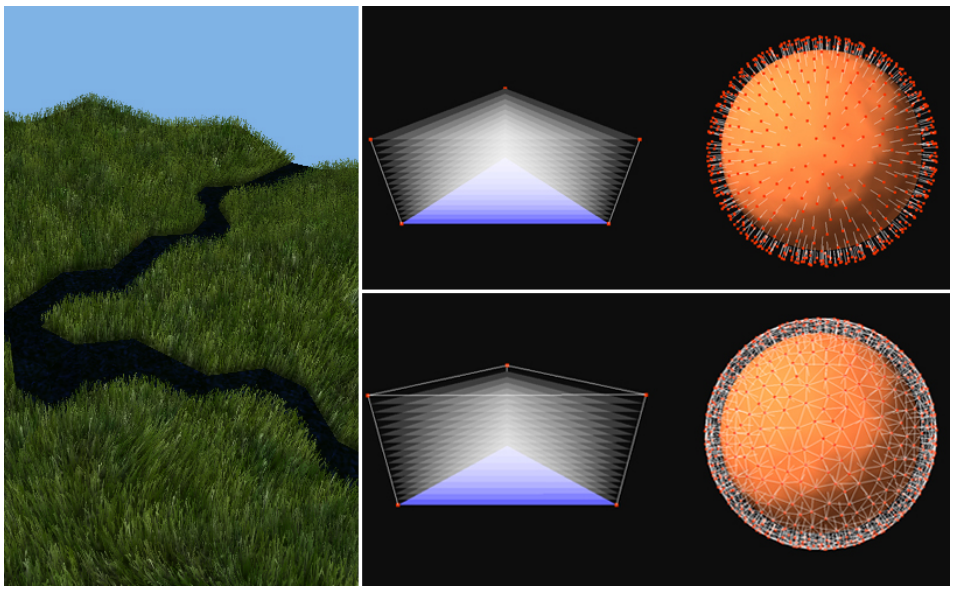

Abbildung 8: Volumenbasiertes Renderingverfahren mittels Shells – hier das Verfahren von [Ban06] mit zwei Topologien am Beispiel einer *Texcell* in der Mitte (oben: *Spring-stick topology*, unten: *Prism topology*)

- *Spring-stick topology*: Bei jedem Vertex der initialen Oberfläche wird je ein Massepunkt an der untersten und einer an der obersten Shell Ebene befestigt. Beide Massepunkte sind mit einer Feder verbunden.
- *Prism topology*: Die Generierung der Massepunkte ist dieselbe wie bei der *spring-stick topology*. Zusätzlich zur vertikalen Verbindung der Massepunkte werden diese auf der obersten Ebene auch horizontal mit Federn verbunden.
- *Seperable spring-sticks* und *separable prism*: Damit sich das Gras in Teilen voneinander unabhängig seitlich in verschiedene Richtungen bewegen kann, werden entlang der Dreiecke zwischen den Vertices zusätzliche Massepunkte und Federn generiert.

Diese Topologien werden in einem Vorverarbeitungsschritt erzeugt. Zur Laufzeit wirken dann Kräfte (z. B. Wind und Schwerkraft) auf die Massepunkte, deren Positionen durch einen Feder-Masse-Löser (engl. solver) berechnet werden. Trotz des physikalischen Modells ist diese Methode echtzeitfähig und liefert für globale Kräfte wie Wind ansehnliche Ergebnisse.

3.2 Einzelne Gräser (Kollisionen)

Noch schwieriger als die Animation der gesamten Wiese ist die Animation einzelner Gräser. Daher beherrschen die hier vorgestellten Systeme auch

die einfachere Bewegung des gesamten Grases. Die Anzahl an Publikationen in diesem Bereich ist noch recht überschaubar und soll durch diese Diplomarbeit um einen weiteren Ansatz bereichert werden.

3.2.1 Prozedurale Methoden

Das älteste Verfahren, um spezifische Grashalme zu bewegen, stammt von Guerraz [GPR+03] und ist eine Erweiterung des Verfahrens von Perbet [PC01], der zur Simulation von Wind sogenannte Windprimitive einsetzt (siehe Kapitel 3.1.1). Für das Kollisionsverhalten des Grases mit Objekten wird das Verfahren um ein Trittprimitiv (treading primitive) ergänzt. Dieses Animationsprimitiv wird an jedes bewegliche Objekt, wie z. B. dem Fuß eines Charakters, fixiert und bestimmt, wie das Gras verbogen und zerdrückt wird und wie lange es benötigt um seine initiale Position wiederzugewinnen.

Die aktuellste Arbeit im Bereich von interaktiv reagierendem Gras stammt von Chen und Johan [CJ10]. Das Gras wird durch Billboards repräsentiert und auf Grund von Gras-Objekt, Gras-Gras, Gras-Wind und Gras-Terrain Interaktionen animiert. Wind bzw. Objekte verursachen eine Reaktion auf zunächst direkt anliegendes Gras. Diese Grasbüschel propagieren den Kontakt an umliegendes Gras weiter, womit eine Art wellenförmige Ausbreitung entsteht. Am Ende wird sich das Gras von der Kollision erholen und zum Ausgangszustand zurückkehren. Um diese Art an Ausbreitung zu erreichen, wird die Wiese als ein Kontinuum modelliert. Die Interaktionen zwischen den Gräsern stellen sich dann als Energietransfer zwischen den Gräsern in diesem Kontinuum dar. Als numerisches Modell für die Dynamik des Kontinuums dient eine Wellensimulation. Das Resultat ihrer Arbeit ist eine Simulation mit Kollisionserkennung und -behandlung mit einer Wellensimulation.

3.2.2 Physikalische Simulationen

Giacomo [GCF01] verwendet zum Animieren von mehreren Bäumen ein Level-of-Detail-System. Einerseits nutzt er für Wind die Windprimitive von Perbet, also eine prozedurale Methode, andererseits verwendet er für das genauere Animieren einzelner Äste ein physikalisches Modell. Hierzu werden die benötigten Daten für die Physiksimulation (z. B. Masse, Kraft, Steifheit) dynamisch anhand der geometrischen Daten des Baumes von einer prozeduralen Methode generiert. Wann welches Animationsverfahren verwendet wird, kann individuell eingestellt werden, beispielsweise anhand des Abstands zum Betrachter – eine Überblendung zwischen beiden Verfahren findet statt um Animationssprünge zu verhindern.

Orthmann [ORSK09], der Billboards zum Darstellen seiner Grasbüschel verwendet, nutzt zum einen für die Illusion von Wind eine einfache Ap-

proximation der Sinusfunktion um die oberen Vertices seiner Billboards zu verschieben. Zum anderen verwendet er für die Kollisionsreaktion des Grases das Konzept von Federn aus Physiksimulationen. Zunächst wird eine grobe Kollisionserkennung auf der CPU durchgeführt und die möglichen Kollisionen an die Grafikkarte (GPU) übergeben, um den genauen Kollisionstest, die Kollisionsbehandlung und die Erholungsphase (das Zurückbiegen nach der Kollision) des Grases zu berechnen. Wird eine Kollision zwischen Gras und Objekt erkannt, so werden die Positionen der Vertices des Grasbillboards aus dem Objekt herausbewegt. Zusätzliche Vertices werden, sofern eine Kollision erkannt wurde, dazu auf dem Texturquad erzeugt. Damit dabei die allgemeine Form des Grasbüschels erhalten bleibt, kommt ein Stoffmodell auf Basis eines Feder-Systems zum Einsatz. Wird eine Feder stark gestaucht oder gestreckt, so sorgt sie dafür, dass die verbundenen Vertices entsprechend mitbewegt werden, gleichzeitig auf Grund der Steifheit der Feder aber nicht zu sehr auseinandergezerrt werden.

4 Konzeptfindung und -idee

Auf Grund der Tatsache, dass die Kollisionsbehandlung von Gras mit Objekten in bisherigen Veröffentlichungen oft keine Rolle spielten, war in dieser Diplomarbeit früh klar, dass eine eigene Idee verwirklicht werden sollte. Eine interaktive und eigens gestaltbare virtuelle Welt ist immer öfter ein entscheidendes Kriterium bei Computerspielen. Erst seit wenigen Jahren ist die Verarbeitungsgeschwindigkeit von Computern so schnell, dass auch dieser Faktor bei der Darstellung von Gras mit einfließen kann. Daher wird im Folgenden dargelegt, welche Anforderungen es zu bewältigen galt und welches Konzept daraus entstand.

4.1 Anforderungen

Auf Grund der Anforderung einer Echtzeitsimulation ließen sich schon frühzeitig in der Konzeptphase viele Visualisierungstechniken ausschließen. Echtzeitsimulationen werden als solche mit minimal 15 Frames pro Sekunde wahrgenommen (S. 1, [AMHH08]). Bei 30 Frames pro Sekunde kann man sicher von Echtzeit, und damit von flüssiger Bilddarstellung, reden. 6 Frames pro Sekunde (fps) reichen für die sogenannte interaktive Benutzung.

Trotz der Einschränkung auf die Echtzeitfähigkeit des Systems soll es ein glaubhaftes Ergebnis liefern, das mit der Realität vergleichbar ist. Da Gras nicht als einzelner Halm sondern als gesamte Wiese wahrgenommen wird, muss der entstehende Prototyp große Grasfelder darstellen können und möglichst nicht auf eine bestimmte Größe beschränkt sein.

Einen elementaren Bestand dieser Diplomarbeit stellt die Kollisionserkennung und -behandlung dar. Unterschiedliche Objekte sollen damit eine

Reaktion des Grases auslösen. Neben dem Einfluss von Kollisionen auf das Gras zählt auch der äußere Einfluss des Winds eine entscheidende Rolle auf die wahrgenommene Realitätsnähe. Daher darf auch der Wind in dieser Simulation nicht fehlen.

Da einzelne Grashalme oder auch Grasbüschel verglichen mit der Größe der gesamten Wiese verhältnismäßig klein sind, sollte als Blickwinkel bzw. Blickpunkt der eines Menschen oder kleiner gewählt werden. Feine Kollisionen im Gras würden aus der Vogelperspektive nicht zur Geltung kommen. Auch in der Realität sehen wir kleinste Bewegungen im Gras, vor allem wenn dieses kurz ist, nur aus nächster Nähe. Entsprechend soll die Visualisierungskomponente des Prototypen für ein ansprechendes Ergebnis aus der *First Person* Perspektive sorgen, da auch aus dieser die Kollisionen erkennbar sind.

Zu weiteren erforderlichen Eigenschaften der Software sollte die freie Bedienung der Kamera und der Kollisionsobjekte zählen um die Funktionalität der Kollisionserkennung prüfen zu können.

4.2 Konzeptidee

Im Folgenden werden die wichtigsten Bausteine des Konzepts erläutert und ihre Verwendung begründet. Eine detailliertere Beschreibung findet sich in Kapitel III.

4.2.1 Visualisierung

Renderingverfahren

Zum Darstellen der Gräser soll ein bildbasiertes Verfahren zum Einsatz kommen. Geometriebasierte Verfahren sind für ein Echtzeitsystem zu langsam und wären nur in Zusammenhang mit einem Level-of-Detail-System sinnvoll verwendbar. Da allerdings bildbasierte Verfahren ähnlich detaillierte Objekte simulieren können und eine aufwändige Überblendung bei einem Level-of-Detail-System hiermit entfällt, ist kein großer Nachteil zu befürchten. Zudem werden Texturquads bereits erfolgreich in den aktuellsten Computerspielen verwendet um Gras darzustellen. Auf diese Quads werden semi-transparente Texturen gelegt. Mit ihnen können sehr große Rasenflächen versehen werden mit trotzdem guten Bildwiederholraten. Volumenbasierte Verfahren scheiden aus, da die Animation und insbesondere die Kollisionserkennung dieser nur sehr aufwändig zu lösen wären.

Level-of-Detail-System

Level-of-Detail-Verfahren bieten bei korrekter Verwendung einen enormen Geschwindigkeitsvorteil. Mit ihnen ist eine praktisch unendlich große Graslandschaft realisierbar mit dabei vorhersehbarer Berechnungskomplexität.

Außerdem entspricht ein solches System eher unserer Wahrnehmung der Realität: Weit entfernte Wiesen erscheinen uns als eine Farbe und einzelne Gräser sind nicht mehr zu erkennen. Gras, welches sich hingegen direkt vor uns befindet, ist im Detail sichtbar. Auch Bewegungen und Kollisionen sind in der Ferne nicht erkennbar oder höchstens über eine Verfärbung des verbogenen Grases auszumachen. Ohne Level-of-Detail-System wäre hingegen die Vorberechnung des Grases wesentlich einfacher möglich und zusätzliche Rechenschritte auf Grund der Berechnung der Detailstufen zur Laufzeit könnten vermieden werden. Eine erste Implementation sollte daher zunächst auch ohne Level-of-Detail-System funktionieren und um dieses später erweitert werden. Damit können unterschiedliche Bedürfnisse bedient werden: Bei kleinen Wiesen kann auf eine aufwändige Level of Detail Berechnung zur Laufzeit verzichtet werden. Benötigt man hingegen eine größere Wiese, so greift das Level-of-Detail-System und kann die Graslandschaft ohne weiteres darstellen. Der erste und zweite Level of Detail sollte aus den oben genannten Texturquads bestehen. Der zweite Level besteht aus einer reduzierten Anzahl an Quads. Der dritte Level ist eine einfache Bodentextur.

Beleuchtung

Für die Beleuchtung der Graslandschaft kommt eine texturbasierte Beleuchtung mittels der für das Gras verwendeten Textur zur Anwendung. Da die Beleuchtung statisch sein wird, kann mit einer einfachen Verschattung innerhalb der Textur gearbeitet werden. Weitere Effekte sind dank der verwendeten einfachen Geometrie (Texturquads) möglich. Die Verwendung von Screen Space Ambient Occlusion (siehe Kapitel 8.7.1) kann die Schattenbildung verbessern.

4.2.2 Animation

Da sich die wenigsten Verfahren, die das Animieren der gesamten Wiese übernehmen, auf das Animieren einzelner Grashalme auf Grund von Kollisionen anwenden lassen, wird folgend auf eine getrennte Betrachtung verzichtet und stattdessen direkt das Problem einzelner Grashalme betrachtet. Aufbauend auf der Entscheidung ein bildbasiertes Verfahren mit möglicher Verwendung eines Level-of-Detail-Systems zur Darstellung des Grases einzusetzen, muss ein Verfahren gefunden werden, das mit diesen Gegebenheiten funktioniert.

Da eine Kollisionserkennung und -behandlung bei aktuellen Physik-Engines[13] bereits eingebaut ist, schien die Nutzung eines physikbasierten Systems eine sinnvolle Vorgehensweise zu sein.

[13] Programmteil zum Lösen mathematischer Formeln zur Simulation physikalischer Gegebenheiten

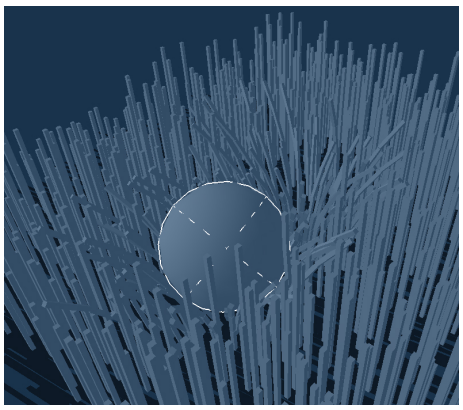

Abbildung 9: Nvidia PhysX (Beispiel aus dem SDK, *Lesson405*)

Zunächst wurde eine dem Nvidia PhysX SDK beiliegende Beispielapplikation untersucht, die einzelne Grashalme mittels Starrkörpern und Gelenken realisiert. Diese sehr genaue Simulation funktioniert bei wenigen Grashalmen sehr gut und läuft auf der CPU flüssig. Bei der üblichen Größe von Wiesen ist dieses Verfahren allerdings viel zu langsam. Nvidia PhysX berechnet die Starrkörpersimulation komplett auf der CPU und bietet hierfür keine GPU-Beschleunigung an. Bei wenigen tausend Starrkörpern bzw. Constraints (zu deutsch Zwangsbedingungen; hier Bedingungen, die die freie Bewegung der Starrkörper einschränken) sind heutige Prozessoren nach wie vor überfordert, wie sich auch anhand der von Nvidia mitgelieferten Beispielanwendung nachvollziehen ließ. Nach genauerer Recherche stellte sich heraus, dass Nvidia PhysX maximal 64.000 Aktoren berechnen kann. Für mehr Aktoren muss die Szene in mehrere Unterszenen unterteilt werden, was ein aufwändiges Verfahren darstellt. Obwohl das Beispiel des Nvidia PhysX Software Development Kits explizit eine Grassimulation darstellt, ist diese Vorgehensweise nicht auf eine größere Wiese übertragbar. Auf Grund dieser Erfahrungen wurde dieser Ansatz für eine weitere Betrachtung verworfen.

Anstelle der genauen Nachmodellierung von Grashalmen mittels Starrkörpern kommt für einen physikbasierten Ansatz nur noch eine Approximation in Frage. Partikel (siehe auch Kapitel 5.1) stellen eine solche Annäherung dar und lassen sich bereits heute mit Hilfe der GPU beschleunigen. Partikelsysteme kommen bereits bei vielen Effekten und Problemen zum Tragen, wie z. B. Feuer, Rauch aber auch bei Haarsimulationen, die, wie eingangs erwähnt, wiederum gewisse Ähnlichkeiten zu Grassimulationen besitzen. Partikelsysteme werden von 3D-Animations- und Visualisierungsprogrammen wie Blender[14] verwendet, um Gras für Filme zu generieren und zu animieren. Daraus ergab sich die Idee, das Gras in Echtzeit mit Hilfe

[14] http://www.blender.org/

eines Partikelsystems zu animieren. Jedes Texturquad wird dabei von einem Partikel repräsentiert, welches mittels eines Constraints an der Wurzelstelle des Grasbüschels befestigt ist. Jedes Grasquad wird dann anhand der Position des Partikels verschoben. Da Partikelsysteme mit Millionen an Partikeln (siehe [Lat04]) bereits in Echtzeit berechnet werden können, kann angenommen werden, dass ein solches auch für die Grassimulation von tausenden an Grasbüscheln nutzbar ist. Eine mögliche Beschleunigung via Grafikkarte ist ebenfalls vorstellbar. Die Kollisionserkennung zwischen Grasbüschel und Objekten soll zwischen den Partikeln und den Starrkörpern stattfinden und die veränderten Partikelpositionen werden an die Grasbüschel weiter propagiert. Das Level-of-Detail-System zur Visualisierung des Grases wird auch für das Physiksystem aufgegriffen und die Physik nur in der ersten, dem Betrachter nächsten, Detailstufe berechnet. Bewegungen von Gras, vor allem von kurzem Gras, können vom menschlichen Auge auch in der Realität nur im Nahbereich erkannt werden. Diese Beobachtung wird mit dem Level-of-Detail-System auf die Virtualität übertragen. Ebenso wird mit Wind verfahren, welcher über Kräfte, die auf die Partikel wirken, gesteuert wird.

Teil II
Theoretische Grundlagen

Im Folgenden werden die Grundlagen von Physik-Engines erläutert, da eine solche für die Animation des Grases verwendet und erweitert wird. Eine Physik-Engine berechnet die Mathematik zum Simulieren von physikalischen Prozessen. Für Echtzeitanwendungen wird die Physik approximativ dargestellt bzw. berechnet, da eine exakte Simulation der Realität zu aufwändig wäre. Bestandteile einer solchen Physik-Engine können beispielsweise sein:

- Starrkörper-Simulation (Rigid Bodies Simulation)
- Simulation deformierbarer Körper (Soft Bodies Dynamics)
- Masse-Feder-Simulation (Mass-Spring Dynamics), z. B. für Stoffe
- Partikelsysteme (Particle System) z. B. für Flüssigkeiten oder Rauch
- Kollisionserkennung (Collision Detection)
- Kollisionsbehandlung (Collision Response)

Da in dieser Diplomarbeit Partikel zur Animation des Grases herangezogen werden und Starrkörper mit Gräsern kollidieren können, werden die Starrkörpersimulationen, Partikelsysteme, Kollisionserkennung und -behandlung nun weiter betrachtet. Die Grundlagen und Formeln sind aus [Mil07] entnommen – für weitere Details, die über den Umfang dieser Diplomarbeit hinausreichen, wird diese Literatur zum Nachschlagen empfohlen.

5 Physiksimulationen

Zunächst wird die einfachste Größe – das Partikel – einer Physik-Engine behandelt. Physikalische Gesetze, die für ein Partikel gelten, müssen auch für Starrkörper beachtet werden. Für Starrkörper müssen einige zusätzliche Eigenschaften berücksichtigt werden.

5.1 Partikelsystem

5.1.1 Größen eines Partikels und physikalische Gesetze

Ein Partikelsystem zeichnet sich meistens durch eine große Anzahl an Partikeln aus. Ein Partikel, oder auch Massepunkt, wird durch eine Position im Raum bestimmt. Es hat keine räumliche Ausdehnung und damit einhergehend auch keine Orientierung, was zugleich den größten Unterschied zum

Starrkörper darstellt. Weitere wichtige Parameter eines Partikels sind seine Masse (mass m), Geschwindigkeit (velocity $\vec{v}$), Beschleunigung (acceleration $\vec{a}$) und Kräfte (force $\vec{f}$), die auf das Partikel wirken. Manche Größen können aus anderen berechnet werden – sinnvoll ist die separate Speicherung dennoch, um erneute Berechnungen zu vermeiden. Hierzu werden die newtonschen Gesetze der Physik benötigt.

Das erste newtonsche Gesetz sagt aus, dass ein Körper in Ruhe verweilt oder sich mit konstanter Geschwindigkeit bewegt, wenn die Summe der auf ihn wirkenden Kräfte gleich null ist. Das heißt, dass sich die Position des Partikels entweder nicht verändert oder auf Grund der Geschwindigkeit ändert. In der Realität werden Objekte auf Grund von Kräften wie z. B. Reibung dennoch irgendwann zum Stehen kommen. Solche Kräfte können ebenfalls berücksichtigt werden – im einfachsten Fall durch sukzessive Subtraktion von Geschwindigkeit.

Das zweite newtonsche Gesetz beinhaltet den Zusammenhang zwischen Kraft, Masse und Beschleunigung:

$$\vec{f} = m \cdot \vec{a} \tag{1}$$

$$\vec{a} = \frac{1}{m} \cdot \vec{f} \tag{2}$$

Eine Kraft $\vec{f}$ ändert also in Abhängigkeit von der Masse m des Körpers die Beschleunigung $\vec{a}$ dieses Körpers. In einer Physiksimulation werden Kräfte verwendet um die Position eines Objekts zu ändern. Eine bekannte und in jeder Physiksimulation vorhandene Kraft ist die Gravitationskraft, die durch die Formel

$$\vec{f} = m \cdot g \tag{3}$$

beschrieben wird, wobei $g = 9,81 \frac{m}{s^2}$ die Erdanziehungskraft ist. Eingesetzt in Gleichung 2 ergibt dies $\vec{a} = g$, d. h. die Beschleunigung ist konstant. Gravitation wirkt unabhängig von der Masse und kann deshalb als konstante Beschleunigung (Vektor mit negativer y-Komponente) realisiert werden ohne über die Gleichung der Kraft 1 zu gehen.

5.1.2 Die Integration

Zur Umsetzung der im vorherigen Kapitel vorgestellten Gesetze benötigt die Physik-Engine einen Integrator. Dieser ist dafür zuständig, in jedem Frame die neuen Positionen und Geschwindigkeiten aller Partikel zu berechnen.

Zu einer Zeit $t = 0$ befindet sich ein Körper beispielsweise an Position $\vec{p}$, zur Zeit $t = 1$ an Position $\vec{p}'$. Die mittlere Geschwindigkeit lässt sich mittels $\vec{v} = \frac{\vec{p}' - \vec{p}}{\Delta t}$ ermitteln. Für einen bestimmten Zeitpunkt ergibt sich

$$\vec{v} = \frac{d\vec{p}}{dt} = \dot{p} \tag{4}$$

Die Punktschreibweise bedeutet somit, dass sich die Position $\vec{p}$ mit der Geschwindigkeit $\vec{v}$ ändert, also $\vec{v}$ das (erste) Differential von $\vec{p}$ in Bezug auf die Zeit ist.

Mit der Beschleunigung wird analog verfahren: Sie wird mittels der Formel $\vec{a} = \frac{d\vec{v}}{dt}$ berechnet und drückt aus, wie sich die Geschwindigkeit mit der Zeit ändert. Setzt man dies in Formel 4 ein, erhält man

$$\vec{a} = \frac{d}{dt}\frac{d\vec{p}}{dt} = \frac{d^2\vec{p}}{dt^2} = \ddot{p} \tag{5}$$

und damit $\vec{a}$ als das zweite Differential.

Wenn wir die Geschwindigkeit $\vec{v}$ bzw. die Beschleunigung $\vec{a}$ eines Körpers kennen, so können wir die Position $\vec{p}$ bzw. Geschwindigkeit $\vec{v}$ zu einem späteren Zeitpunkt bestimmen. Dazu müssen wir die obigen Formeln integrieren. Für die neue Position eines Objekts ergibt dies

$$\vec{p}' = \vec{p} + \dot{p}t \tag{6}$$

und für die neue Geschwindigkeit

$$\dot{p}' = \dot{p} + \ddot{p}t \tag{7}$$

Zum Lösen der Formeln benötigt es noch ein Zeitintervall – hierzu dient beispielsweise die Zeitspanne zwischen zwei Frames. Die Integrationsfunktion einer jeden Physik-Engine belebt also überhaupt erst die in der Szene enthaltenen Körper, indem sie laufend die aktuellen Positionen und Geschwindigkeiten der Partikel ermittelt. Für Starrkörper müssen diese Formeln erweitert werden.

5.1.3 Constraints

Constraints, übersetzt Zwangsbedingungen oder Beschränkungen, dienen dazu die Bewegungsmöglichkeiten eines Partikels einzuschränken. Im Rahmen dieser Diplomarbeit werden sogenannte *harte* Constraints benötigt um die Partikel an einer Stelle zu halten. Konkret handelt es sich dabei um einen Ankerpunkt, an dem das Partikel hängt, welches immer in einem festgelegten Abstand zu diesem gehalten wird. Ein Constraint kann als eine spezielle Art einer Kollision gesehen werden, da ein Verstoß wie eine Kollision gelöst werden muss. Millington [Mil07] spricht in diesem Zusammenhang von einem *Kontakt*. Der Kollisionsbehandlungsmethode muss dazu der Anker und die einzuhaltende Distanz von diesem mitgeteilt werden. In Kapitel 7 wird auf das Lösen von Kollisionen eingegangen.

5.2 Starrkörpersimulation

Starrkörper, im Englischen Rigid Bodies, unterscheiden sich von Massepunkten durch ihr festes Volumen. Dies macht es nötig Rotationen zu betrachten, welche die Mathematik von Starrkörpern deutlich komplexer werden lässt.

5.2.1 Rotationen und verwandte Größen

Die Orientierung eines Objekts wird durch eine Rotationen relativ zum Ursprung eines Objekts geändert. Es gibt verschiedene Möglichkeiten Rotationen in drei Dimensionen anzugeben:

- Rotationsmatrix: Matrix, die die Lage der drei lokalen (orthonormalen) Koordinatenachsen im Raum beinhaltet
- Eulerwinkel: Drei Winkel, die die jeweilige Rotation um drei Achsen beschreiben
- Axis-Angle: Repräsentation einer Rotation mittels einer Achse, um die mit einem Winkel rotiert wird
- Quaternionen: Darstellung der Rotationsachse und des Rotationswinkels in einem Vektor mit vier Werten, wobei hierfür eine eigene Algebra benötigt wird. Ein Quaternion schreibt man $(cos\frac{\theta}{2}, x \cdot sin\frac{\theta}{2}, y \cdot sin\frac{\theta}{2}, z \cdot sin\frac{\theta}{2})$, wobei (x, y, z) die Achse und θ der Winkel sind.

Sie lassen sich ineinander überführen. Zusätzlich zur eigentlichen Orientierung muss gespeichert werden, wie schnell sich Körper in eine bestimmte Richtung drehen (Winkelgeschwindigkeit $\dot{\theta}$, engl. angular velocity) und mit welcher Beschleunigung (Winkelbeschleunigung $\ddot{\theta}$, engl. angular acceleration).

5.2.2 Das zweite newtonsche Gesetz für Starrkörper

Befindet sich der Ursprung des Körpers an dessen Schwerpunkt, so verhält sich die lineare Bewegung des Objekts wie die eines Partikels. Ist dies nicht der Fall oder tritt eine externe Kraft (wirkt auf den Mantel des Körpers und nicht auf seinen Schwerpunkt) auf, so können infolgedessen Rotationen auftreten, die bei den Berechnungen berücksichtigt werden müssen.

Ähnlich zu Formel 2 (wobei $\vec{a} = \ddot{p}$ nach 5) für die lineare Geschwindigkeitsänderung, kann auch die Winkelbeschleunigung berechnet werden:

$$\ddot{\theta} = I^{-1}\vec{\tau} \tag{8}$$

I, das Trägheitsmoment (auch Inertialmoment, Massetensor), engl. moment of inertia, ist das winkelabhängige Äquivalent zur Masse. Ebenso stellt

τ, die Drehkraft (auch Drehmoment), engl. torque, ungefähr das Gegenstück zur Kraft dar.

Das Trägheitsmoment beschreibt, wie schwierig es ist die Drehgeschwindigkeit des Objekts zu ändern. Besitzt ein Gegenstand eine große Masse und möchte man ihn zum Rotieren bringen, so muss man eine größere Kraft aufwenden, umso weiter der Punkt der Kraftanwendung von der Drehachse entfernt ist. Da das Trägheitsmoment von der Drehachse abhängig ist, wird es in Physik-Engines als *Tensor*, eine „lineare Funktion zwischen Vektoren, die jedem Vektor einen anderen zuordnet“ [Mül10], repräsentiert. Für einige geometrische Formen, wie z. B. für die Kugel oder einen Würfel, gibt es einfache Formeln zum Berechnen ihres Tensors. Dies ist der Grund, wieso in Physik-Engines alle Berechnungen auf primitive Körper zurückgeführt werden.

Die Drehkraft tritt infolge einer Kraft, die nicht auf den Schwerpunkt des Körpers wirkt, auf. Der Körper wird sich drehen mit

$$\vec{\tau} = \vec{p}_f \times \vec{f}, \tag{9}$$

wobei $\vec{p}_f$ der Ort der Kraftausübung relativ zum Ursprung des Körpers und $\vec{f}$ die eigentliche Kraft sind.

Die Integration von Starrkörpern findet nun analog zu der von Partikeln statt. Zusätzlich wird die Orientierung des Körpers berücksichtigt und die Winkelgeschwindigkeit (um eine Achse) ähnlich zur linearen Geschwindigkeit (Formel 7) mittels

$$\dot{\theta}' = \dot{\theta} + \ddot{\theta}t \tag{10}$$

berechnet und aktualisiert.

5.2.3 Kugel und Würfel

Für die in dieser Diplomarbeit verwendete und erweiterte Physik-Engine werden die zwei wichtigsten Starrkörper Kugel und Würfel berücksichtigt.

Der Tensor einer Kugel ist

$$I = \begin{bmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{bmatrix} \tag{11}$$

wobei m seine Masse und r der Radius ist.

Für einen Quader gilt

$$I = \begin{bmatrix} \frac{1}{12}m(d_y^2 + d_z^2) & 0 & 0 \\ 0 & \frac{1}{12}m(d_x^2 + d_z^2) & 0 \\ 0 & 0 & \frac{1}{12}m(d_x^2 + d_y^2) \end{bmatrix} \tag{12}$$

mit m als Masse und d_x, d_y und d_z als Ausmaße entlang der Achsen des Quaders. Ein Würfel ist als Spezialfall des Quaders damit leicht zu bestimmen.

Gespeichert werden jeweils die inversen Tensoren, die mittels Matrixinvertierung bestimmbar sind.

6 Kollisionserkennung

Ohne Kollisionserkennung würde das Gras auf keine Objekte reagieren. In komplexen Szenen können sich hunderte oder tausende von Objekten befinden. Die Kollisionserkennung muss jedes Objekt mit jedem Objekt nach möglichen Kollisionen (im Weiteren auch Kontakt genannt) untersuchen. Dies entspricht n^2 Tests. Um diese Menge zu reduzieren wird die Kollisionserkennung in zwei Phasen unterteilt:

1. **Die grobe Kollisionserkennung**
   (coarse oder broad collision detection)
   Zunächst soll die Menge der n^2 Berechnungen reduziert werden, da die Kalkulation aller n^2 Kollisionen zu viel Zeit benötigen würde. Ein Objekt am einen Ende der Szene muss nicht auf Kollision mit einem Grashalm am anderen Ende getestet werden. Daher wird die Szene bzw. ihre Objekte mittels spezieller Datenstrukturen unterteilt und auf Grund dieser die Anzahl der möglicherweise kollidierenden Objekte verringert.

2. **Die genaue Kollisionserkennung**
   (fine oder narrow collision detection)
   Nachdem die Menge der möglicherweise kollidierenden Objekte durch die grobe Kollisionserkennung erstellt wurde, prüft die genaue Kollisionserkennung nun, ob es sich um tatsächliche Kollisionen handelt und berechnet die für die Kollisionsbehandlung benötigten Daten (wo und wie sich die Körper berühren).

Auf beide Abschnitte der Kollisionserkennung wird im Weiteren genauer eingegangen.

6.1 Die grobe Kollisionserkennung

Die grobe Kollisionserkennung soll die Menge möglicher Kollisionen berechnen. Unter diesen möglichen Kontakten müssen sich alle tatsächlichen Kollisionen befinden, es können sich aber auch solche befinden, die in Wirklichkeit keine sind (sogenannte *false positives*). Die gesamte grobe Kollisionserkennung muss möglichst schnell verlaufen und sollte im Optimalfall wenige false positives enthalten.

Um die Berechnung zu beschleunigen werden oftmals Bounding Volumes[15] verwendet. Hierfür dienen geometrische Primitive, die die eigentlich darzustellenden und zu simulierenden Objekte umschließen. Kugeln sind die einfachsten Bounding Volumes, da sie durch einen Punkt und einen Radius leicht zu beschreiben sind und zudem rotationssymmetrisch sind. Auch Würfel lassen sich ähnlich intuitiv durch einen Mittelpunkt und seine (halbe) Kantenlänge beschreiben.

In Zusammenhang mit diesen Bounding Volumes werden meistens sogenannte Bounding Volume Hierarchien verwendet. Übergeordnete Bounding Volumes werden generiert, die zwei oder mehr Bounding Volumes enthalten. Diese zusätzlichen Volumen werden in eine Baumstruktur eingetragen, wobei die Blätter des Baums die einzelnen Bounding Volumes sind und der Wurzelknoten ist ein Bounding Volume, das alle anderen enthält. Dadurch ist es möglich Kollisionstests auf einer größeren Bounding Volume durchzuführen und bei negativem Ergebnis auf weitere Tests der kleineren beinhalteten Bounding Volumes zu verzichten. Obgleich eine zusätzliche Datenstruktur benötigt wird, können möglicherweise viele Objekte und deren Bounding Volumes aus der Kollisionsberechnung ausgeschlossen werden.

Eine weitere Möglichkeit der Reduzierung der Tests besteht in der Nutzung von räumlichen Datenstrukturen (spatial data structures). Teilweise werden auch in diesen Datenstrukturen die Bounding Volumes verwendet oder Hierarchien durch zusätzliche räumliche Datenstrukturen erweitert. Unter räumliche Datenstrukturen fallen die binäre Raumpartitionierung (Binary Space Partitioning, BSP), Quadtrees, Octrees und Gitter (Grids). Für diese Diplomarbeit besonders interessant ist die Gitterstruktur, da das Gras bereits in einer Gitterstruktur platziert ist. Dadurch ist es möglich direkt festzustellen, welche der Grasbüschel von den Kollisionsobjekten möglicherweise berührt werden, da anhand der Positionen der Objekte direkt auf die Partikelpositionen geschlossen werden kann. Abhängig von der Position der Objekte werden dann nur die nächsten Partikel in der ausführlicheren zweiten Phase der Kollisionserkennung berücksichtigt.

6.2 Die genaue Kollisionserkennung

Sind durch die grobe Kollisionserkennung die möglichen Kandidaten ermittelt worden, so liegt es nun an der genauen Kollisionserkennung die tatsächlich kollidierenden Objekte zu erfassen und die für die spätere Kollisionsbehandlung benötigten Daten zu berechnen.

Anstatt die für die Ausgabe konzipierten hochauflösenden Körper mit mehreren tausend Polygonen für die Kollisionserkennung zu verwenden, wird eine separate Kollisionsgeometrie benutzt. Dazu werden oftmals meh-

[15] geometrischer Körper, der einen komplexeren Körper umschließt

rere unterschiedliche Geometrieprimitive aneinander gehängt. Je akkurater die grobe Geometrie für die zu simulierenden Körper gewählt werden, desto akkurater und passender ist auch das eigentliche Ergebnis für diesen Körper. Die Nutzung einer Kugel als approximierter Körper für eine Kiste würde beispielsweise bedeuten, dass sich die Kiste wie eine Kugel verhält. Alle echtzeitfähigen Physik-Engines bauen auf vereinfachten Geometrien auf um die Integration und die Kollisionsberechnungen zu vereinfachen.

Für diese Diplomarbeit werden die Kugel und der Würfel als Kollisionsgeometrien verwendet um die prinzipielle Einsetzbarkeit einer Physik-Engine für die Grassimulation bewerten zu können. Eine Erweiterung um andere Primitive und auch deren Kombinationen ist grundsätzlich möglich.

Die Kollision zwischen Partikeln und Starrkörpern ist für die Grassimulation besonders wichtig. Für den Eindruck einer funktionierenden Physik werden auch Kollisionen zwischen Starrkörpern und zwischen Starrkörper und Terrain behandelt. Weiterhin muss das Partikelconstraint (siehe 5.1.3) mittels Kollisionserkennungsalgorithmus gelöst werden: Es liefert ähnliche Daten zurück wie gewöhnliche Kollisionen und wird auf Grund der geringeren Rechenkomplexität (keine Starrkörper und damit keine Rotationen) während der Kollisionsbehandlung separat behandelt.

6.2.1 Kollisionen zwischen Partikeln und Starrkörpern

Die Kollisionserkennung von Partikel mit Starrkörper ist insofern einfach, da ein Partikel keine Ausdehnung bestitzt. Die Kollisionserkennung wird meistens erst dann eine Kollision erkennen, wenn das Partikel bereits in den Körper eingedrungen ist und nicht genau dann, wenn die Oberfläche berührt wird. Eine Kollision zwischen Kugel und Partikel findet tatsächlich statt, wenn gilt: $radius_{sphere} > \sqrt{(\vec{p}_{sphere} - \vec{p}_{particle})^2} > 0$. Für Würfel und Partikel muss gelten: $halfSizeEdge_{cube} - \vec{p}_{particle} \geq 0$.

Nun müssen folgende Daten erfasst werden (siehe auch Abbildung 10):

- **Kontaktpunkt**: Dies ist der Punkt, an dem sich die Körper berühren. In diesem Fall ist dies die Position des Partikels $\vec{p}_{contact} = \vec{p}_{particle}$.
- **Kollisionsnormale**: Dies ist der Vektor, in dessen Richtung sich beide Objekte in der Kollisionsbehandlung auseinander bewegen werden. Für die Kollision von Partikel mit Kugel ist dies der Vektor zwischen Kugelmittelpunkt und Partikelposition. Für die Kollision von Partikel mit Würfel hat sich gezeigt, dass eine konstante Normale von $\vec{n} = (0, 1, 0)$ den besten visuellen Eindruck hinterlässt, wenngleich dies nicht physikalisch korrekt ist. Partikel werden dadurch nach unten durch das Terrain gedrückt und benötigen so länger, um zu ihrer Ausgangsposition zurückzukehren. Dies entspricht genau dem Verhalten von Gras, welches eine gewisse Zeit benötigt um in seine Ausgangslage zurückzukehren.

- **Eindringtiefe**: Die gegenseitige Durchdringung wird in der Eindringtiefe (engl. penetration) festgehalten. Sie wird ausgehend vom Kontaktpunkt in Richtung der Kollisionsnormalen gemessen. Für die Kollision mit einer Kugel berechnet sich dies durch $penetration = radius_{sphere} - \sqrt{(\vec{p}_{sphere} - \vec{p}_{particle})^2}$. Bei einem Würfel wird ähnlich einfach gerechnet: $penetration = halfSizeEdge_{cube} - \vec{p}_{particle}$. Bei der allgemeineren Form des Quaders (unterschiedliche Kantenlänge je Achse) kann diese Formel ebenfalls verwendet werden, wenngleich die kleinste Eindringtiefe gespeichert werden muss.

6.2.2 Kontaktgenerierung Partikelconstraints

Eine zu lösende Kollision, hier besser als Kontakt bezeichnet, entsteht bei dem Partikelconstraint dann, wenn die Länge des Constraintvektors (Vektor vom Ankerpunkt zum Partikel) die vorgeschriebene Länge verletzt. Als zu übergebene Parameter werden dann festgehalten:

- der jeweilige Ankerpunkt,
- als Kontaktnormale der Vektor zwischen Partikel und Anker,
- als Eindringungstiefe die momentane Länge des Constraints.

6.2.3 Sonstige Kollisionen

Wie bei der Kollision zwischen Partikel und Starrkörper müssen die Variablen Kontaktpunkt, Kollisionsnormale und Eindringtiefe auch für die übrigen Kollisionstypen berechnet werden.

Für die Daten der Kugel-Kugel und Kugel-Box Kollisionen wurden die folgenden Formeln direkt aus [Mil07] übernommen.

Kugel-Kugel Kollision

Wenn die Distanz zwischen den Kugelmittepunkten kleiner ist als deren summierte Radien, dann kollidieren die Kugeln miteinander. Für die Variablen gilt:

- **Kontaktpunkt**: Der Punkt liegt auf dem Verbindungsvektor beider Kugelmittelpunkte: $\vec{p}_{contact} = \vec{p}_{sphere1} + \frac{1}{2}\sqrt{(\vec{p}_{sphere1} - \vec{p}_{sphere2})^2}$.
- **Kollisionsnormale**: Die Normale $\vec{n} = \frac{(\vec{p}_{sphere1} - \vec{p}_{sphere2})}{\sqrt{(\vec{p}_{sphere1} - \vec{p}_{sphere2})^2}}$ entspricht dem Vektor zwischen den Kugelzentren.
- **Eindringtiefe**: Die Eindringtiefe berechnet sich aus der Differenz zwischen den addierten Radien beider Vektoren und der tatsächlichen Distanz zwischen den Positionen der Kugeln: $penetration = radius_{sphere1} + radius_{sphere2} - \sqrt{(\vec{p}_{sphere1} - \vec{p}_{sphere2})^2}$.

Kugel-Würfel Kollision

Es gibt drei unterschiedliche Kontaktarten zwischen Kugel und Würfel: Die Kugel kann auf einer Kante des Würfels aufkommen, auf einer Fläche und auf einer Ecke. Ausgehend von der Kugel können alle drei Fälle gleich berechnet werden.

- **Kontaktpunkt**: Der Kugelmittelpunkt wird zuerst in das Objektkoordinatensystem des Würfels transformiert und ist damit relativ zur Orientierung des Würfels. Für den Kontaktpunkt wird dann angenommen, dass er auf den Flächen des Würfels liegt und er erhält die Koordinaten der $halfSizeEdge_{cube}$ in denjenigen vom Würfel ausgehenden drei Achsenrichtungen ($+$ oder $-$), die die minimale Distanz zur Kugelposition besitzen.
- **Kollisionsnormale**: Für die Normale gilt aus der Berechnung des Kontaktpunkts $\vec{p}_{contact}$ dann: $\vec{n} = \vec{p}_{sphere} - \vec{p}_{contact}$.
- **Eindringtiefe**: Die Eindringtiefe ergibt sich aus $penetration = radius_{sphere} - \frac{\vec{p}_{contact} - \vec{p}_{sphere}}{\sqrt{(\vec{p}_{contact} - \vec{p}_{sphere})^2}}$.

Starrkörper-Terrain Kollision

Auch für die Kollision zwischen den Starrkörpern mit dem Terrain müssen die obigen Daten erfasst werden. Die (interpolierte) Höhe und die (interpolierte) Normale des Terrains sind gegeben. Daraus folgt:

- **Kontaktpunkt**: Für den Kontaktpunkt einer Kugel mit dem Terrain muss der Lotfußpunkt eines Punktes (Kugelmittelpunkt) zu einer Ebene (Terrain) gefunden werden. Es gilt $\vec{p}_{contact} = \vec{p}_{sphere} - \vec{n} \cdot e$, wobei $\vec{n}$ die Normale des Terrains an der auf das Terrain projizierten Kugelposition ist und $e = \vec{n} \cdot \vec{p}_{sphere} - d$ mit $d = \vec{n} \cdot \vec{p}_{terrain}$.
  Für die Kollision eines Würfels mit dem Terrain wurde der in [Mil07] vorhandene Kollisionsfall Würfel mit einer Ebene (genauer Halbraum) modifiziert. Jeder Eckpunkt des Würfels wird separat mit dem Terrain auf Abstand getestet um zu bestimmen, auf welcher Würfelseite der Würfel auf dem Boden liegt. Diese vier Punkte werden als Kontaktpunkte gespeichert.
- **Kollisionsnormale**: Als Normale für die Kugel- und Würfelkollision wird die Terrainnormale verwendet (siehe Kontaktpunkt).
- **Eindringtiefe**: Für die Eindringtiefe bei der Kollision einer Kugel mit dem Terrain gilt: $penetration = \vec{p}_{sphere} \cdot (0,1,0) - radius_{sphere} - height_{terrain}$.
  Für den Würfel gilt wiederum: $penetration = height_{terrain} - \vec{p}_{cubecorner} \cdot (0,1,0)$.

Partikel-Terrain Kollision

Auf eine Kollisionserkennung von Partikel mit dem Terrain wurde bewusst verzichtet, da das Eindringen der Partikel in das Terrain eine visuelle Erholungsphase der Gräser darstellt. Da ein Partikel, welches um $180°$ in das Terrain verdrängt wurde, eine längere Strecke zu seiner Ausgangsposition zurücklegen muss, ist dies gleichbedeutend mit einer längeren Phase des Liegens eines Grasbüschels. Unangenehme Artefakte in der Darstellung entstehen dadurch nicht.

7 Kollisionsbehandlung

Nach der Generierung der Kontaktdaten durch die Kollisionserkennung kann nun die Kollisionsbehandlung einsetzen und die Kollisionen lösen. Damit Objekte voneinander abprallen, muss die Geschwindigkeit der Objekte geändert werden. Die in dieser Diplomarbeit zugrunde liegende Physik-Engine nutzt einen Impuls-basierten Ansatz um diese Änderung der Geschwindigkeit auszuführen. Ein Körper, der auf dem Terrain liegt, erhält in jedem Frame einen Impuls, der den Körper ein kleines Stückchen vom Terrain wegstößt. Eine andere Möglichkeit besteht darin, eine Kraft über eine kleine Zeitspanne auf einen solchen Körper auszuüben (Kraft-basierter Ansatz). Diese Methode soll laut [Mil07] allerdings eine komplexere Mathematik voraussetzen, weshalb hier nur die Grundlagen des Impuls-basierten Ansatzes vorgestellt werden.

7.1 Geschwindigkeitsänderung

Aufbauend auf den Newtonschen Gesetzen müssen die linearen und winkelabhängigen Geschwindigkeiten geändert werden. Für die sofortige lineare Geschwindigkeitsänderung wird ein Impuls benötigt, für die winkelabhängige Geschwindigkeitsänderung ein Drehkraftimpuls (engl. impulsive torque).

Für den Impuls $\vec{g}$ gilt

$$\vec{g} = m\dot{p} \tag{13}$$

und entsprechend für den Drehkraftimpuls $\vec{u}$

$$\vec{u} = I\dot{\theta} \tag{14}$$

Beide Formeln sind angelehnt an die Formeln für die Kraft $\vec{f} = m\ddot{p}$ (siehe Formel 1) und die Drehkraft $\vec{\tau} = I\ddot{\theta}$ (siehe Formel 8). Beide Impulse verhalten sich dementsprechend wie Kräfte. Eine besondere Schwierigkeit ist die Bestimmung des Verhältnisses von Impuls und Drehkraftimpuls für

die Objekte. Für zwei aneinander stoßende Körper werden vier Impulse benötigt, für Kollisionen auf Grund des Terrains nur zwei.

Zum Berechnen dieser Impulse sind folgende Schritte notwendig:

1. Da die Kollisionsbehandlung in einem eigenen Koordinatensystem arbeitet, welches relativ zum Kontaktpunkt liegt, muss eine Transformationsmatrix erstellt werden um zwischen den Koordinatensystemen umrechnen zu können (Kapitel 7.1.1).

2. Berechnung der Geschwindigkeitsänderung am Kontaktpunkt für beide Objekte (Kapitel 7.1.2).

   - Um eine Geschwindigkeitsänderung zu erreichen wird ein Impuls benötigt. Durch den Impuls ergibt sich eine lineare und eine winkelabhängige Bewegung.
   - Wie groß die Änderung der Geschwindigkeit sein soll, ergibt sich aus der gewünschten Geschwindigkeit, wenn sich die Körper voneinander wegbewegen (Geschwindigkeit nach dem Kontakt), und der momentanen Geschwindigkeit, wenn sich die Körper zueinander hinbewegen (Geschwindigkeit vor dem Kontakt).

3. Mittels der in Schritt 2 berechneten gewünschten Änderung der Geschwindigkeit kann der dazu benötigte Impuls generiert werden. Der Impuls wird in einen linearen und einen winkelabhängigen Teil aufgetrennt und auf beide Objekte angewendet (Kapitel 7.1.3).

7.1.1 Das Kontakt-Koordinatensystem

Um die folgende Mathematik zu vereinfachen, werden die Berechnungen relativ zum Kontaktpunkt gelöst (siehe auch Abbildung 10). Das Koordinatensystem hat seinen Ursprung im Kontaktpunkt. Die x-Achse ist die bereits vorhandene Kontaktnormale. Die y-Achse und z-Achse können mehr oder weniger frei gewählt werden, müssen aber auf jeden Fall eine orthonormale Basis[16] bilden.

Um zwischen Kontakt- und Weltkoordinatensystem hin und her zu transformieren, wird eine Transformationsmatrix $M_{c \to w}$ benötigt. Sie lässt sich einfach konstruieren, indem die Achsen des Kontaktkoordinatensystems als Spalten in die Matrix eingesetzt werden:

$$x_c = \begin{pmatrix} a \\ b \\ c \end{pmatrix}, \; y_c = \begin{pmatrix} d \\ e \\ f \end{pmatrix}, \; z_c = \begin{pmatrix} g \\ h \\ i \end{pmatrix}$$

[16] Menge von Vektoren, die den Vektorraum erzeugen, und die normiert und zueinander orthogonal sind[Wik10a]

$$M_{c\to w} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

Mit $M_{c\to w}\cdot\vec{p}_c = \vec{p}_w$ kann dann ein Punkt vom Kontaktkoordinatensystem in das Weltkoordinatensystem umgerechnet werden. Für die umgekehrte Richtung gilt $M_{w\to c} = M_{c\to w}^T$.

7.1.2 Die gewünschte Geschwindigkeitsänderung

Da wir die Geschwindigkeit eines Punktes auf einem Objekt berechnen müssen, benötigen wir folgende Formel, die eine winkelabhängige und eine lineare Komponente besitzt

$$\dot{q} \quad = \quad \underbrace{\dot{\theta} \times (\vec{q} - \vec{p})}_{winkelabhängig} + \underbrace{\dot{p}}_{linear}$$

mit $\vec{q}$ als Position des Punktes auf dem Objekt in Weltkoordinaten, $\vec{p}$ als Position des Objekts, $\dot{q}$ und $\dot{p}$ ihre jeweiligen Geschwindigkeiten und $\dot{\theta}$ die winkelabhängige Geschwindigkeit des Objekts.

Die lineare Änderung der Geschwindigkeit für einen Einheitsimpuls in dessen Richtung berechnet sich durch $\Delta\dot{p} = m^{-1}$ bzw. für zwei kollidierende Objekte durch

$$\Delta\dot{p} = m_a^{-1} + m_b^{-1}$$

mit m^{-1} als die jeweilige inverse Masse des Körpers. Für die winkelabhängige Änderung der Geschwindigkeit für einen Drehkraftimpuls gilt

$$\Delta\dot{\theta} = I^{-1}\vec{u} \tag{15}$$

mit

$$\vec{u} = \vec{q}_{rel} \times \vec{n}_{contact} \tag{16}$$

wobei $\vec{q}_{rel} = \vec{q} - \vec{p}$ (also die Position relativ zum Objektursprung) ist und $\vec{n}_{contact}$ die Kontaktnormale, die gleich der Richtung des Impulses ist.

Um nun zu berechnen, wie groß die gewünschte voneinander wegbewegende Geschwindigkeitsänderung sein soll, benötigt man die Formel $\vec{v}_s' = -c\vec{v}_s \Rightarrow \triangle\vec{v}_s = -\vec{v}_s - c\vec{v}_s$ mit $\vec{v}_s$ als die relative Geschwindigkeit der Objekte vor bzw. $\vec{v}_s'$ nach der Kollision und c der Restitutionskoeffizient[17]. Die neue Geschwindigkeit wird in entgegengesetzter Richtung wirken.

[17] Stoßzahl, um eine Mischform aus ideal elastischem und ideal plastischem Stoß darzustellen [Wik10d]

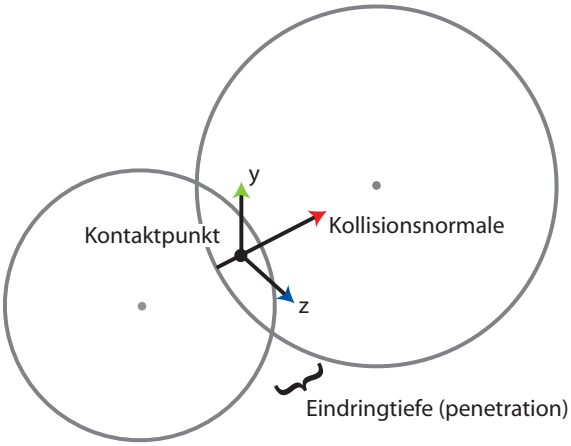

Abbildung 10: Kontakt-Koordinatensysteme und Vektoren (nach [Mil07])

7.1.3 Anwendung des Impulses

Für Kontakte ohne Reibung ergibt sich für den Impuls $\vec{g}$, der eine gegebene Geschwindigkeitsänderung $\vec{v}$ erzeugt (siehe Kapitel 7.1.2), die Formel $\vec{g} = \frac{\vec{v}}{d}$, wobei d die Geschwindigkeitsänderung pro Einheitsimpuls ist. Dieser Impuls wirkt in Richtung der Kontaktnormalen und kann mit Hilfe der Matrix $M_{c \to w}$ (siehe Kapitel 7.1.1) in Weltkoordinaten zurückkonvertiert werden. Für die lineare Geschwindigkeitsänderung gilt nach Formel 13 $\dot{p} = \frac{\vec{g}}{m}$ und für den winkelabhängigen Teil nach Formel 15 $\Delta\dot{\theta} = I^{-1}\vec{u}$, nachdem mit Formel 16 der Drehkraftimpuls berechnet wurde. Für den ersten Körper können die Ergebnisse direkt verwendet werden, für den zweiten Körper muss der Impuls in die entgegengesetzte Richtung wirken, weshalb die Berechnungen für diesen erneut durchgeführt werden müssen.

Falls Reibung vorhanden ist, wird sich ein Objekt auf Grund dessen irgendwann nicht mehr bewegen, d. h. man muss die Bewegung eines Objekts zusätzlich vermindern und zwar in Richtung der Bewegung. Die in diesem Kapitel vorgestellten Formeln müssten entsprechend erweitert werden, da bisher eine Änderung der Geschwindigkeit nur in Richtung der Kontaktnormalen vollzogen wurde. Da diese Kontakte mit Reibungen nicht das eigentliche Augenmerk für das in dieser Diplomarbeit gestellte Problem darstellen, sei hier auf die Literatur verwiesen.

7.2 Positionsänderung

Da die Kollisionsbehandlung nur einmal pro Zeitschritt (hier pro Frame) aufgerufen wird, können sich Gegenstände in der Zeit dazwischen durchdringen ehe eine Kollision erkannt wird. Daher müssen diese Überschneidungen durch konkrete Positionsänderungen der festen Körper gelöst werden. Hierzu wurde in der Kollisionserkennungsphase die Eindringtiefe gespeichert.

Um die neuen Positionen zu bestimmen, gibt es verschiedene Möglichkeiten, von denen die folgenden zwei für diese Grassimulation von Bedeutung sind:

- Lineare Projektion: Die Positionen der Objekte werden entlang der Kontaktnormalen um die minimale Distanz, bis sich beide Objekte nicht mehr berühren, bewegt. Dieses Vorgehen ist einfach zu implementieren und wird für das Partikelconstraint verwendet, da bei diesem keine Rotationen berücksichtigt werden müssen. Diese führen dazu, dass eine lineare Projektion auf Starrkörper unrealistisch wirken würde. Die nichtlineare Projektion ist für Starrkörper besser geeignet.
- Nichtlineare Projektion: Bei der nichtlinearen Projektion wird eine Kombination von linearer und winkelabhängiger Bewegung verwendet um die Eindringung zu lösen. Wie bei der linearen Projektion werden die Körper entlang der Kontaktnormalen voneinander wegbewegt mit dem Unterschied, dass der (inverse) Trägheitstensor beider Objekte für die Berechnung der Bewegung mit einbezogen wird.

7.2.1 Lineare Projektion

Da die Angaben der Eindringtiefe und der Kontaktnormalen bereits vorhanden sind, muss nun noch bestimmt werden, um wieviel jeder einzelne Körper bewegt werden muss. Bei einer Kollision zwischen Starrkörper und Terrain muss der Körper um die komplette Eindringtiefe hinausbewegt werden. Bei einer Kollision zwischen zwei Körpern wird die anteilige Distanz anhand der Masse der Körper bestimmt. Ein leichter Körper wird dadurch mehr bewegt als ein schwerer. Für beide Positionsänderungen gilt $\Delta\vec{p}_a + \Delta\vec{p}_b = penetration$. Mit der Massebedingung ergibt sich dann

$$\begin{aligned} \Delta\vec{p}_a &= \frac{m_b}{m_a + m_b} \cdot penetration \cdot \vec{n} \\ \triangle\vec{p}_b &= -\frac{m_a}{m_a + m_b} \cdot penetration \cdot \vec{n} \end{aligned}$$

Für Partikelkollisionen (hier das Partikelconstraint) kann auf die Massebedingung verzichtet werden, da die Masse aller Partikel gleich ist.

7.2.2 Nichtlineare Projektion

Für die nichtlineare Projektion müssen die linearen und winkelabhängigen Komponenten der Projektionsbewegung bestimmt werden. Dazu wird ein Trägheitsfaktor i (wie schwierig es ist einen Körper zu bewegen) bestimmt, wobei dessen linearer Teil einfach die inverse Masse $i_{linear} = m^{-1}$ des Körpers ist. Für den winkelabhängigen Anteil gilt

$$i_{angular} = I^{-1} \cdot (\vec{q}_{rel} \times \vec{n}_{contact}) \cdot \vec{n}_{contact}.$$

Die Anwendung dieses Faktors für die Positionsänderung geschieht im Falle des linearen Anteils direkt über die Multiplikation mit der Kontaktnormalen und im Falle des winkelabhängigen Anteils über die Multiplikation dieses Faktors mit einer Rotation, die den Kontaktpunkt um eine Einheit bewegt. Diese Rotation wird in Anlehnung an die Impulsrechnungen (siehe Formel 15) berechnet, da im Zusammenhang mit Impulsen der Einheitsimpuls eine ähnliche Bedeutung innehat.

Teil III
Praktische Umsetzung

Ziel dieser Diplomarbeit soll eine „prototypische Echtzeitanwendung sein, die eine ansprechende Darstellung und Physiksimulation von Gras bietet“. Diese wurde nach dem Konzept aus Kapitel 4 implementiert. Entwickelt wurde unter Windows 7 x64 mit Microsoft Visual Studio 2008 in C++, OpenGL 2.0 und GLSL.

Die folgenden Frameworks bzw. Bibliotheken wurden verwendet um zusätzliche Funktionalitäten zu gewährleisten:

- GLFW: Dieses Toolkit wird verwendet, um den OpenGL Rendering Kontext zu erstellen und Tastatureingaben und Mauseingaben zu verarbeiten. GLFW unterstützt Multisampling, welches für Alpha-To-Coverage (siehe Kapitel 8.7.2) benötigt wird.
- GLUT: Das OpenGL Utility Toolkit wird ausschließlich für die Darstellung einfacher (Debugging-)Modelle (z. B. Teapot, WireFrameSphere) verwendet um keine Inteferenzen mit GLFW zu erzeugen.
- GLEW: Die OpenGL Extension Wrangler Library ermöglicht das einfache Laden und Verwenden zusätzlicher OpenGL Extensions.
- DevIL: Mit der Developer's Image Library können jegliche Art von Bildern geladen werden und direkt in OpenGL eingebunden werden.
- AntTweakBar: Wird verwendet um eine OpenGL basierte GUI bereitzustellen.
- Cyclon Physik-Engine: Auf Basis dieser Engine wurde die Physiksimulation integriert und erweitert. Trotz anfänglicher Tests mit der Nvidia PhysX-Engine fiel die Entscheidung auf die Nutzung einer Physik-Engine, die im Detail anpassbar ist. Die (iterative, impulsbasierte) Cyclon Engine wird in [Mil07] entwickelt und beschrieben. Im Gegensatz dazu ist die PhysX-Engine eine fertige Physik-Engine, deren Partikelsystem oder Kollisionserkennung beispielsweise nicht den eigenen Bedürfnissen angepasst werden können.

Weiterhin wurden diese Assets bzw. Klassen verwendet um kleinere Funktionen bereitzustellen:

- Nate Robins GLM : Diese Klasse ermöglicht das Laden von OBJ-Modellen.

- Imageloader für Heightmaps[18]: Um spezifische Terrains laden zu können, wird diese einfache Klasse verwendet um BMP Dateien zu laden. Diese beinhalten die Höheninformationen für die Terraindarstellung.
- dhpoware Terrain Texturing[19]: Um das Terrain mitsamt Texturen darzustellen, wurde diese Demoimplementation verwendet, die mehrere Texturen mittels GLSL-Shader überblendet.
- Echtzeitrendering Framework: Für die Vorlesung Echtzeitrendering wurde von der Computergraphik AG ein Framework zur Verfügung gestellt, das einige Grundfunktionalitäten in Klassen zusammenfasst. Einzelne Klassen wie z. B. eine `Time` oder `FrameBufferObject` Klasse wurden übernommen und ggf. angepasst.

In den weiteren Kapiteln werden hinsichtlich der Visualisierung und der Physiksimulation Details der Implementation, genannt *SimGrass*, beschrieben. Insbesondere wird darauf hingewiesen, welche Möglichkeiten bestanden um die Performance der Darstellung und die Darstellungsqualität zu verbessern und welche Strategien entwickelt wurden um die Kollisionserkennung und -behandlung zu beschleunigen.

8 Visualisierung

Auf Grund des verwendeten Level-of-Detail-Systems (siehe Kapitel 8.3) unterteilt sich die Visualisierung in drei unterschiedliche Grasrepräsentationen:

- Gras
- Kantengras
- Grasbodentextur

Dabei unterscheidet sich das Gras vom Kantengras durch die verwendete Datenstruktur und damit einhergehend durch die eingesetzten Aktualisierungs- und Generierungsmethoden. Daher wird ihre Implementation getrennt betrachtet.

Sie haben allerdings auch einige Gemeinsamkeiten, wie z. B. die Verwendung von Vertex Buffer Objects und Texturatlanten. Die Nutzung von Screen Space Ambient Occlusion als Post-Processing Effekt ist ebenfalls unabhängig von dieser Aufteilung.

[18] http://www.videotutorialsrock.com/opengl_tutorial/terrain/home.php
[19] http://www.dhpoware.com/demos/glslTerrainTexturing.html

8.1 Geometrie rendern

Um Geometrie in OpenGL zu zeichnen gibt es verschiedene Möglichkeiten:

- Immediate Mode
- Display Lists
- Vertex Arrays
- Vertex Buffer Objects (VBO)

Der Immediate Mode ist die mit Abstand langsamste Methode und sollte daher nur für einfache Ausgaben z. B. zu Debuggingzwecken verwendet werden. Display Lists werden einmalig erstellt und können dann nicht mehr geändert werden. Somit sind sie für die Darstellung animierten Grases nicht geeignet. Bei Vertex Arrays wird auf die Vertexdaten, Normalen, Farben etc., die im Hauptspeicher liegen, mittels Zeiger zugegriffen. Im Unterschied dazu werden beim VBO diese Daten in Buffer Objects gespeichert, die von OpenGL allokiert werden und im schnellen Grafikkartenspeicher gehalten werden. Ab OpenGL Version 3.0 sind alle Methoden bis auf die Vertex Buffer Objects *deprecated*, da alle neueren Versionen Vertexdaten auf der Clientseite nicht mehr erlauben. Für die Grasdarstellung wird auf Vertex Buffer Objects zurückgegriffen, da sie auch für die Zukunft die einzige Variante zur Darstellung von Geometrie darstellen werden und zugleich sehr schnell sind. Auch andere Grassimulationen, wie diejenige von Orthmann [ORSK09], nutzen diese Vertex Buffer Objects.

Nach [OW10] sollen separate VBOs verwendet werden, wenn die verwendeten Daten wie die Vertexpositionen dynamisch sind, d. h. pro Frame einmal aktualisiert werden. Da sich die Graspositionen auf Grund von Wind ständig ändern und sich auch die Texturkoordinaten bei Kollisionen ändern (siehe Kapitel 8.2), werden drei separate VBOs erstellt:

- Vertex Buffer (Vertexpositionen)
- Texture Coordinate Buffer (Texturkoordinaten)
- Normal Buffer (Normalen pro Vertex)

Diese drei Buffer werden auch für das Kantengras erstellt, obgleich sich ihre Vertexpositionen und Texturkoordinaten nicht zwangsläufig jedes Frame ändern (dies ist nur der Fall bei Bewegung der Kamera, siehe Kapitel 8.3). Da sich der Performanceverlust bei Nutzung separater VBOs in Grenzen hält (laut [OW10] beträgt dieser ca. 5%), wurde hier auf die Zusammenlegung aller Daten in einen VBO verzichtet.

Die Vertexdaten werden als `GL_QUADS` mittels `glDrawArrays` gezeichnet. Es werden mitunter mehrere zehntausend Quads gezeichnet je nach Größe der Wiese. Die Quads werden im Weiteren texturiert.

8.2 Texturierung

Die eigentlichen Grasbüschel werden mittels semi-transparenter Texturen auf den Quads dargestellt. Die Texturen wurden auf Basis der Grastexturen der Unigine Engine[20] erstellt. Um verschiedene Texturen verwenden zu können, damit die Wiese aus unterschiedlichen Grasbüscheln besteht, gibt es wiederum einige Methoden um diese zu realisieren:

- Nach jedem einzelnen Quad eine neue Textur binden und verwenden: Da VBOs verwendet werden, wäre dieses Verfahren sehr unpraktisch und ist zudem sehr langsam, da der Aufruf von `glBind(GL_TEXTURE_2D)` zu den (in Bezug auf die Performance) teuersten OpenGL-Operationen gehört.
- Verwendung von Texturarrays: Ein `GL_TEXTURE_2D_ARRAY` beinhaltet mehrere Texturen (Layer) gleicher Größe und gleichen Formats. Mittels Shader kann über eine Texturkoordinate und die Layernummer auf eine Textur zugegriffen werden.
- Nutzung eines Texturatlasses: Es wird eine große Textur erstellt, in der unterschiedliche Grasbüschel abgebildet sind. Über unterschiedliche Texturkoordinaten kann dann auf unterschiedliche Grasbüschel zugegriffen werden.

Da ein Texturkoordinaten-VBO verwendet wird, können in ihm direkt die verschiedenen Texturkoordinaten gespeichert werden. Zum Zeitpunkt der ersten Generierung des Grases wird mittels der Texturkoordinaten bestimmt, welches Quad welches Grasbüschel repräsentiert. Texturatlanten haben den großen Vorteil, dass ein einmaliges Binden dieser Textur genügt und trotzdem unterschiedliche Texturen (nämlichen diejenigen innerhalb des Atlasses) verwendet werden können. Zusätzliche Performancekosten entstehen daher nicht. Der verwendete Texturatlas für das Gras ist in Abbildung 22 zu sehen. Er beinhaltet zugleich vier verschiedene Helligkeitsstufen, die für die Neigung des Grases verwendet werden (siehe Kapitel 9.2.6). Zu erkennen ist außerdem auch, dass jede einzelne Textur innerhalb des Atlasses einen gewissen Abstand zu den Nachbartexturen wahren muss, damit auf Grund des verwendeten Mipmappings[21] später keine Farbverfälschungen (Color Bleeding) auftauchen.

Um die transparenten Bereiche des Texturatlasses später auch transparent darzustellen, wird der sogenannte Alphatest verwendet. Dieser muss vor dem Zeichnen der Quads mittels `glEnable(GL_ALPHA_TEST)` aktiviert werden. Zusätzlich muss die Alphafunktion `glAlphaFunc` definiert

[20] http://unigine.com/

[21] Um die Geschwindigkeit zu verbessern und Aliasingeffekte zu verhindern, werden verschiedene Detailstufen einer Textur (die sogenannten Mipmaps) erstellt, die in Abhängigkeit von der Entfernung zum texturierten Objekt verwendet werden.

sein. Mit dieser wird bestimmt, bei welchem Alphawert die Textur transparent dargestellt wird. Mit dem ersten Parameter wird der Alpha-Vergleich festgelegt, in diesem Fall `GL_GREATER` und mit dem zweiten Parameter der Referenzwert, der zwischen 0 und 1 liegt. Ein Wert in der Mitte verhindert, dass unschöne Ränder um das Grasbüschel zu sehen sind.

8.3 Level-of-Detail-System

Die Verwendung eines Level-of-Detail-Systems dient vornehmlich der Performancesteigerung des Systems durch Reduzierung der Vertices. Bei einer guten Überblendung zwischen den Detailstufen ist das Vorhandensein eines solchen Systems nicht erkennbar. Dadurch ist es möglich beliebig große Landschaften mit Gras zu bedecken.

Konkret verwendet diese Grassimulation ein dreistufiges System:

1. um die Kameraposition herum das tatsächlich animierte und beeinflussbare *Gras*
2. um das in Schritt 1 generierte Gras zusätzliches *Kantengras*, das nicht animiert wird
3. für das gesamte Terrain eine *Bodentextur*, die farblich zum vorhandenen Gras passt, damit der Stufenübergang nicht sichtbar ist

Abbildung 11 verdeutlicht den Aufbau des Level-of-Detail-Systems. Das Gras wird in Reihen mit einem festen Abstand (durch Randomisierung wird der Abstand etwas variiert) zueinander erstellt. Auch das Kantengras erhält einen Reihenabstand, wobei dieser größer ist als der für das Gras. Er kann zudem bei Neugenerierung der gesamten Wiese relativ (d. h. als ein Vielfaches) zum Reihenabstand des Grases verändert werden.

Bei der Initialisierung der Grassimulation wird das Gras (und damit auch das Kantengras) mittig zur Kameraposition platziert. Das niedrige Terrain wird mit der Bodentextur versehen. Bei Bewegung der Kamera muss das Gras und das Kantengras entsprechend mitwandern. Dieses Mitwandern ist der entscheidende Teil des Level-of-Detail-Systems. Dabei muss beachtet werden, dass

- die bereits vorhandenen Grasbüschel ihre Positionen beibehalten,
- die Richtung der Kamerabewegung erfasst wird und in diese Richtung neues Gras angefügt wird, gleichzeitig aber auf der entgegengesetzten Seite des Grases Gras auch wieder entfernt wird, damit die Anzahl der Grasbüschel und damit der Vertices stets gleich bleibt,
- die Neigung während der Kamerabewegung beachtet wird und nur dann zusätzliches Gras generiert wird, wenn die auf den Boden projizierte zurückgelegte Strecke größer als die Distanz zwischen den einzelnen Grasreihen ist,

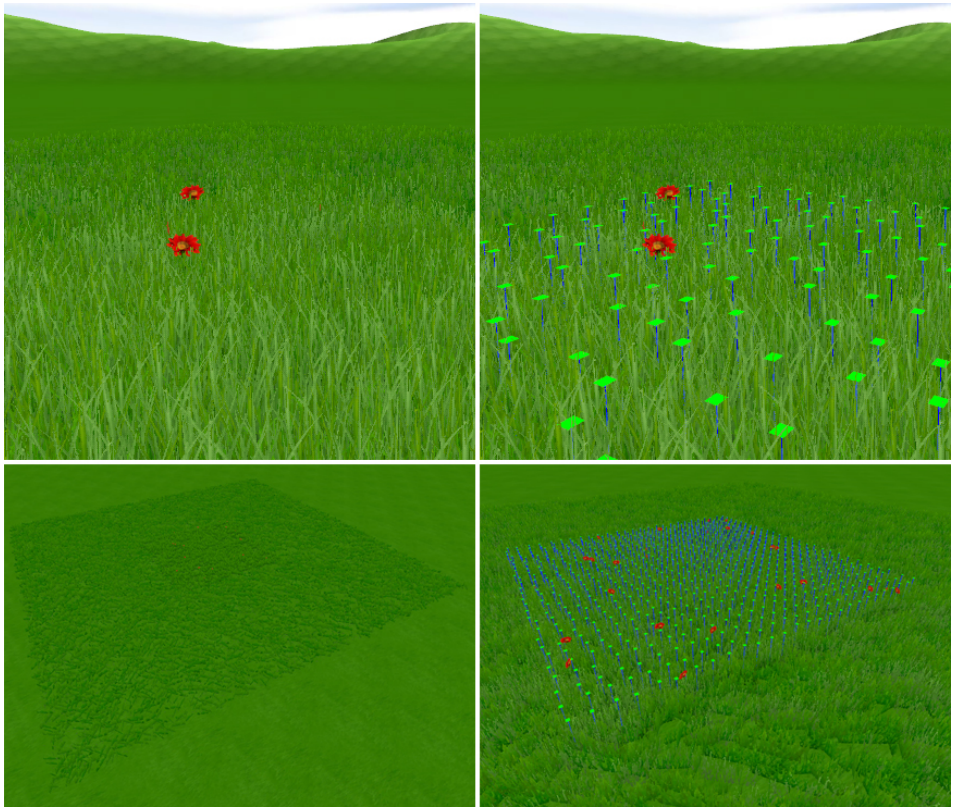

Abbildung 11: Level-of-Detail-System: Oben aus First-Person-Sicht, unten aus der Vogelperspektive (rechts jeweils zur Verdeutlichung mit Anzeige der Partikel)

- möglichst wenige vorhandene Daten kopiert werden müssen, da dies bei den vielen tausend Grasbüscheln eine erhebliche Anzahl wäre,
- die Datenstruktur des Grases so gehalten ist, dass das Partikelsystem darauf einfach abgebildet werden kann.

Da die Anforderungen an das Gras andere sind (zusätzliches Physiksystem) als die für das Kantengras, werden zwei unterschiedliche Datenstrukturen verwendet. Diese werden zusammen mit den für das Level-of-Detail-System benötigten Methoden in den Kapiteln 8.4 und 8.5 getrennt beschrieben. Die für das Physiksystem implementierten Funktionen werden in Kapitel 9 beschrieben. Das Erfassen der zurückgelegten Kamerabewegung ist auf beide Grastypen gleichermaßen anwendbar und wird daher zunächst beschrieben.

8.3.1 Kamerabewegung

Für das eingesetzte Level-of-Detail-System ist es notwendig zu wissen

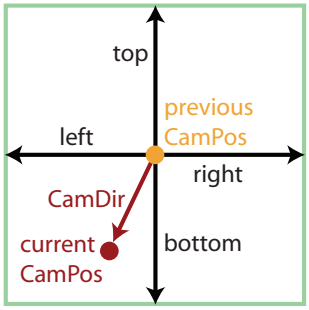

Abbildung 12: Benötigte Vektoren bei Kamerabewegung, wobei $\overrightarrow{side}$ ein Einheitsvektor in die entsprechendene Richtung (top, right, bottom, left) ist.

- in welche Richtung sich der Betrachter, also die Kamera, bewegt und
- welche Distanz dabei zurückgelegt wird.

Mit Hilfe des Skalarprodukts zwischen Kamerarichtung ($\overrightarrow{dir}_{camera} = \vec{p}^{\,current}_{camera} - \vec{p}^{\,previous}_{camera}$) und dem Vektor für die jeweilige Seite ($\overrightarrow{side}$: links, oben, rechts, unten) ist feststellbar ob sich die Kamera in diese Richtung bewegt hat – dies ist der Fall bei einem Skalarprodukt > 0. Abbildung 12 zeigt eine Übersicht über die benötigten Vektoren, die entweder bereits durch das Gras gegeben sind oder einfach berechnet werden können.

Für jede Seite wird die aktuell zurückgelegte Distanz $projectedSide\ Distance$ ausgehend von der Kamerarichtung berechnet

$$\begin{aligned} \overrightarrow{projectedSide} &= \overrightarrow{side} \cdot \frac{\overrightarrow{dir}_{camera} \cdot \overrightarrow{side}}{\overrightarrow{side}.squareMagnitude()} \\ projectedSideDistance &= \overrightarrow{projectedSide}.magnitude() \end{aligned}$$

und im Falle eines positiven Skalarprodukts auf die insgesamt zurückgelegte Distanz in diese Richtung addiert. Auf Grund der Möglichkeit sich völlig frei mit der Kamera in der Simulation zu bewegen, kann man sich in verschiedenen Winkeln zum Terrain bewegen, was diese insgesamt zurückgelegte Distanz nötig macht. Neues Gras soll nur dann generiert werden, wenn sich der Betrachter um die Distanz zwischen den Gräsern in eine Richtung bewegt hat. Hat sich eine insgesamt zurückgelegte Strecke für eine Seite bis auf die Größe des Abstands zwischen den Gräsern akkumuliert, so wird zu diesem Zeitpunkt neues Gras generiert und danach die insgesamte zurückgelegte Distanz um den Abstand wieder verringert. Dies ermöglicht es sich auch schräg zu den Seiten zu bewegen und unterschiedlich viele Grasreihen zu ergänzen, je nachdem, wie sehr sich die Kamera in diese Richtung bewegt.

8.4 Gras

In diesem Kapitel werden jene Daten und Funktionen beschrieben, die für das Gras benötigt werden. Für Funktionalitäten, die in Zusammenhang mit der Animation stehen, sei auf Kapitel 9 verwiesen.

Für das Gras müssen einige Daten gespeichert werden, die grob in zwei STL-Vektoren mit Hilfe von Strukturen (struct) gespeichert werden:

- Positionen: eine x- und z-Weltkoordinate pro Graspatch
- Sonstige Daten pro Graspatch:
  - eine Rotation α mit $[0° \leq \alpha \leq 360°]$
  - Index für die verwendete Textur im Texturatlas
  - der zufällige Offset, der auf die x- und z-Position addiert wird
  - Differenz vom globalen y-Skalierungsfaktor $scaleYMinus$, wenn sich das Gras neigt

Weitere wichtige Daten, die global für alle Graspatches gelten, sind:

- Die Eckpunkte (Weltkoordinaten) des Grasfeldes (aus der Vogelperspektive gesehen links oben, rechts oben, rechts unten und links unten). Diese werden später für das Kantengras benötigt.
- Der Abstand $mSpacing$ zwischen den Graspatches, visuell also wie dicht das Gras beieinander steht.
- Die Anzahl der Graspatches in x- und z-Richtung (besonders wichtig für das Partikelsystem).
- Ein globaler Skalierungsfaktor in y-Richtung $mScaleY$ (Höhe) und xz-Richtung $mScaleXZ$ (Breite).

8.4.1 Generierung

Das Gras kann an einer bestimmten Stelle oder an der aktuellen Kameraposition generiert werden. Nach letzterer Generierung befindet sich die Kamera in der Mitte des Grasfeldes. Für die Startposition des ersten Patches gilt daher:

$$\begin{aligned} startX &= posX_{camera} - spacing \cdot \frac{1}{2} \cdot numberGrassPatchesX \\ startZ &= posZ_{camera} - spacing \cdot \frac{1}{2} \cdot numberGrassPatchesZ \end{aligned}$$

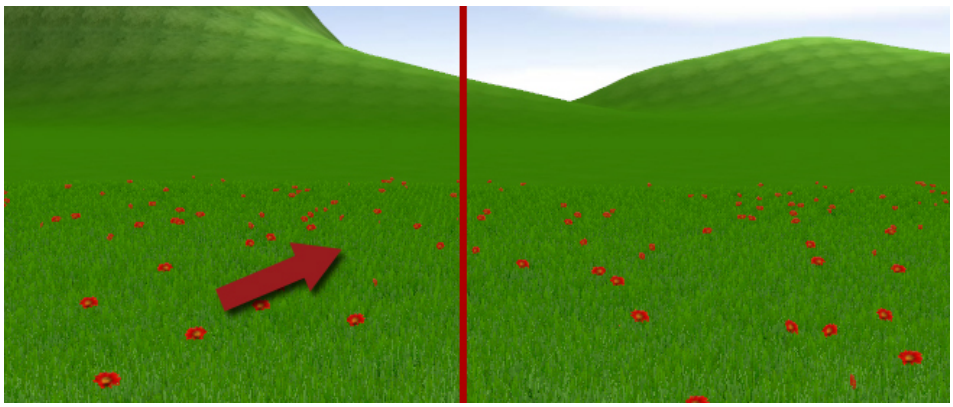

Abbildung 13: Vergleich Graspositionierung ohne (links) und mit (rechts) Zufälligkeitsfaktor

Reihenweise (x-Richtung) werden nun alle Positionen mit dem vordefinierten Abstand $mSpacing$ und der Schleifenvariablen und einem zusätzlichen zufälligen Offset bestimmt. Ohne diesen Offset wäre ein sich wiederholendes Muster erkennbar (siehe Abbildung 13). Zudem werden die übrigen Daten pro Grasbüschel erzeugt, also die Rotation und der Texturatlasindex. Möchte man beispielsweise an bestimmten Positionen eine spezifische Textur anzeigen, so ist dies hier möglich, wie in der Grassimulation mittels einer Fußballfeldmarkierung (siehe Kapitel 11.2) demonstriert wird. Der obige Offset wird ebenso gesichert und die Skalierungsdifferenz $scaleYMinus$ auf den Initialwert 0 gesetzt. Nach der Generierung dieser Daten müssen die Eckpunkte des Grasfeldes gesetzt werden um ausgehend von diesen später das Kantengras erstellen zu können.

Für die Vertex Buffer Objects müssen die Daten noch angepasst werden, ehe sie an die Grafikkarte zur Bildschirmausgabe gesendet werden. Die Daten müssen wie folgt im Speicher liegen:

$$\begin{aligned} V^1_{xyz}V^1_{xyz}V^1_{xyz}V^1_{xyz}V^2_{xyz}V^2_{xyz}V^2_{xyz}V^2_{xyz}... \quad & Vertexpositionen \\ N^1_{xyz}N^1_{xyz}N^1_{xyz}N^1_{xyz}N^2_{xyz}N^2_{xyz}N^2_{xyz}N^2_{xyz}... \quad & Normalen \\ T^1_{st}T^1_{st}T^1_{st}T^1_{st}T^2_{st}T^2_{st}T^2_{st}T^2_{st}... \quad & Texturkoordinaten \end{aligned}$$

mit ^{1,2} als Indizes der Grasbüschel. Auf Grund dieser benötigten sequenziellen Datenstruktur werden diese Daten in drei STL-Vektoren gespeichert, welche direkt an die Grafikkarte übergeben werden können. Für die Vertices eines Grasquads gilt:

$$\begin{pmatrix} -cos_{Rotation} + posX - mScaleXZ \\ terrainheight \\ +sin_{Rotation} + posZ - mScaleXZ \end{pmatrix} \quad Links\ unten$$

$$\begin{pmatrix} +cos_{Rotation} + posX - mScaleXZ \\ terrainheight \\ -sin_{Rotation} + posZ - mScaleXZ \end{pmatrix} \quad Rechts\ unten$$

$$\begin{pmatrix} +cos_{Rotation} + posX - mScaleXZ \\ terrainheight + mScaleY - scaleYMinus \\ -sin_{Rotation} + posZ - mScaleXZ \end{pmatrix} \quad Rechts\ oben$$

$$\begin{pmatrix} -cos_{Rotation} + posX - mScaleXZ \\ terrainheight + mScaleY - scaleYMinus \\ +sin_{Rotation} + posZ - mScaleXZ \end{pmatrix} \quad Links\ oben$$

Sie stehen initial unabhängig von der Steigung des Terrains alle senkrecht zur xz-Ebene des Weltkoordinatensystems auf diesem.

Die Normalen werden pro Vertex angegeben und alle auf $(0, 1, 0)$ gesetzt. Dies zeigte sich für die Beleuchtung als vorteilhaft, da ansonsten möglicherweise Texturquads nur von einer Seite korrekt beleuchtet werden könnten.

Die uv-Texturkoordinaten für den Texturatlas werden unter Angabe des Texturindexes (siehe Abbildung 22) errechnet, wobei in der Gesamttextur links unten $(0.0, 0.0)$ und rechts oben $(1.0, 1.0)$ ist. Für den Index 3 ergeben sich die Texturkoordinaten von links unten ausgehend entgegen des Uhrzeigers z. B. zu $(0.25, 0.5)$,$(0.5, 0.5)$, $(0.5, .075)$ und $(0.25, 0.75)$.

8.4.2 Aktualisierung der Daten bei Kamerabewegung

Wenn sich die Kamera bewegt, müssen unter Umständen neue Grasreihen angefügt und alte gelöscht werden. Ob und wieviele Grasreihen neu hinzugefügt werden müssen, entscheidet die in Kapitel 8.3.1 beschriebene zurückgelegte Distanz der Kamera für jede Seite.

Je nach Seite muss eine andere Vorgehensweise zum Aktualisieren der Grasdaten in den STL-Vektoren genutzt werden, da die Reihenfolge der Daten innerhalb dieser Vektoren wichtig für die Partikelsimulation ist und analog zur Draufsicht auf das Grasfeld beibehalten werden muss. Abbildung 14 zeigt die vier möglichen Fälle zur Aktualisierung der Positionen.

- Fall 1: **Bewegung nach unten**
  Da das Grasfeld (von oben gesehen) reihenweise aufgebaut ist, muss hierbei nur die erste Reihe (also $numberGrassPatchesX$ Einträge) im Vektor gelöscht werden, alle übrigen Reihen im Vektor hinter die neue Reihe gehängt werden und am Ende des Vektors eine neue Reihe angefügt werden.

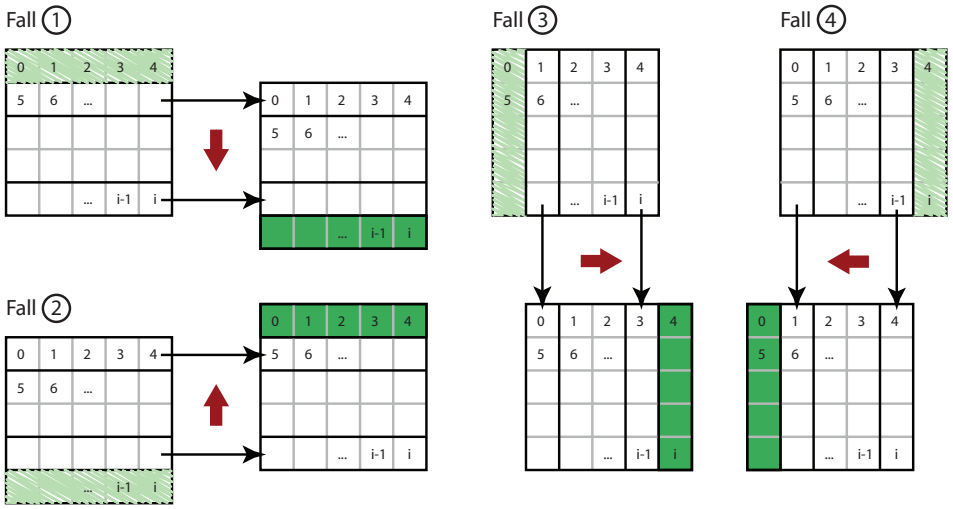

Abbildung 14: Update der STL-Vektoren: Löschen (hellgrün) und Hinzufügen (dunkelgrün) von Grasreihen für die vier Bewegungsfälle

- Fall 2: **Bewegung nach oben**
  Hier muss die neue Reihe an den Anfang des Vektors eingefügt werden und alle übrigen bis auf die letzte Reihe an diese wieder angefügt werden.

- Fall 3: **Bewegung nach rechts**
  Für diese Bewegung geschieht das Update der Grapositionen etwas umständlicher, da hier eine Spalte (im Gegensatz zu Fall 1 und 2 keine Reihe) gelöscht und eine hinzugefügt werden muss. Alle Daten der alten Positionen müssen verschoben um $+1$ im Vektor wieder eingefügt werden.

- Fall 4: **Bewegung nach links**
  Auch für die linke Seite gilt eine spaltenweise, kompliziertere Aktualisierung. Nach Generierung der neuen linken Spalte müssen alle anderen Daten (bis auf die alte rechte Spalte) um -1 im Vektor verschoben übernommen werden.

Im Anschluss müssen die Eckpositionen des Grasfeldes neu gesetzt werden und die STL-Vektoren für die Vertex Buffer Objects ebenfalls entsprechend aktualisiert werden.

8.5 Kantengras

Da das Kantengras nicht animiert wird und abhängig vom Gras um dieses herum generiert wird, kann und muss eine andere Datenstruktur verwendet werden.

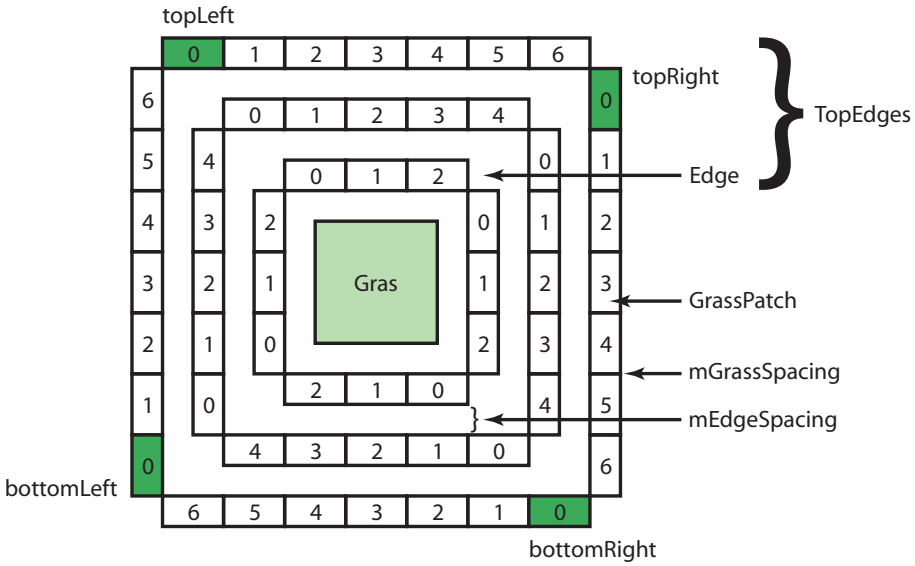

Abbildung 15: Struktur des Kantengrases

Das Kantengras wird in vier Bereiche, die Seiten des Grases, aufgeteilt: links, oben, rechts und unten. Jeder dieser Bereiche, genannt `Edges` und implementiert als STL-Liste, besteht explizit aus mehreren Reihen. Jede Reihe, die `Edge` (ebenfalls eine STL Liste), besteht aus den `GrassPatch`es, die die für die VBOs benötigten Daten (Vertices, Texturkoordinaten, Normalen, zusätzlich auch deren zufälliger x- und z-Offset wie beim Gras) enthalten. Die STL-Listen haben den großen Vorteil, dass nicht nur an das Ende der Datenstruktur neue Daten sehr einfach angefügt werden können, sondern ebenso am Anfang. Dies ist auch notwendig, wie sich in den nachfolgenden Kapiteln zeigen wird. Damit das Kantengras das Gras komplett umschließt, besitzen die einzelnen Reihen des Grases nach außen hin zunehmend unterschiedlich viele Grasbüschel. Abbildung 15 zeigt den Aufbau des Kantengrases.

Neben den eigentlichen Daten für die Grasbüschel besitzt die Klasse `EdgeGrass` anpassbare Variablen, wie z. B. die Anzahl der Reihen, den Abstand der Grasbüschel innerhalb einer Reihe ($mGrassSpacing$) oder den Abstand zwischen den Reihen ($mEdgeSpacing$). Auch das Kantengras besitzt die vom Gras bekannten vier Eckpositionen um schnell zusätzliche Reihen anfügen zu können.

Das Kantengras steht in Abhängigkeit zum Gras, weshalb einige Daten wie z. B. die Höhe des Grases ($mScaleY$) von diesem übernommen werden.

8.5.1 Generierung

Das Kantengras wird in Abhängigkeit von den Eckpunkten des Grases erzeugt.

Die innerste Reihe hat $numberGrassPatchesX + 2 \cdot \frac{mEdgeSpacing}{mGrassSpacing}$ Grasbüschel, damit das Kantengras im $45°$Winkel nach außen hin am Anfang und am Ende breiter wird. Für das erste Patch in der Reihe wird die entsprechende Ecke des Grases als Referenz verwendet. Von dieser wird z. B. im Fall der linken oberen Ecke der Wert von $mGrassSpacing$ und der zufällige Offsetwert dieser Ecke abgezogen um die erste Position (und damit die linke obere Ecke der `EdgeGrass` Klasse) dieser Reihe zu erhalten. Für die anderen Reihen muss analog der $mGrassSpacing$- und Offsetwert addiert bzw. subtrahiert werden.

Sind die Eckpunkte für das Kantengrass so bestimmt worden, kann für jede Seite jeweils die erste Reihe generiert werden. Dies beinhaltet nun noch die Berechnung der tatsächlichen Vertexpositionen, Texturkoordinaten und Normalen, die identisch zur Berechnung dieser für das Gras ist (siehe Kapitel 8.4.1). Alle Patches innerhalb einer Reihe haben einen Abstand von $mGrassSpacing$ zueinander, weshalb nach dem Erzeugen eines Patches die für das nächste Patch benötigte x- bzw. z-Koordinate durch Addieren bzw. Subtrahieren von $mGrassSpacing$ in Richtung der Reihe erhalten werden kann.

Nach Generierung dieser Reihe wird die nächste Eckposition bestimmt. Dazu wird von der aktuellen Eckposition der $mEdgeSpacing$ Wert subtrahiert bzw. addiert. Dies garantiert, dass das Kantengras weiter im $45°$ Winkel versetzt generiert wird und der Anfang einer Reihe einer Seite an das Ende der entsprechenden Reihe der benachbarten Seite nahtlos aneinanderschließt. Die Anzahl der Grasbüschel für die nächste Reihe muss entsprechend um $2 \cdot \frac{mEdgeSpacing}{mGrassSpacing}$ Patches im Vergleich zur Vorgängerreihe erhöht werden. Die äußeren Reihen haben somit immer mehr Grasbüschel als (zum Gras hin) weiter innen liegende. Die eigentliche Reihengenerierung geschieht genauso wie für die allererste Reihe.

Nach dem ersten Generieren des gesamten Kantengrases werden bei Bewegung der Kamera nur noch einzelne Grasbüschel bzw. Reihen ergänzt und gelöscht, was dank der verwendeten Datenstruktur und der STL-Listen syntaktisch einfach zu bewerkstelligen ist.

8.5.2 Aktualisierung der Daten bei Kamerabewegung

Diese durch das Level-of-Detail-System nötige Neugenerierung einiger Teile des Kantengrases gestaltet sich nach Abbildung 16 in mehreren Schritten. Im Beispiel wird sie für die Bewegung des Betrachters nach rechts dargestellt – das Prinzip ist analog aber für jede andere Richtung ebenso anwendbar. Die Richtung ergibt sich wie beim Gras durch das im Kapitel 8.3.1 beschriebene

Verfahren. Da das Kantengras allerdings einen anderen Abstand zwischen den Reihen besitzen kann als das Gras, müssen die zurückgelegten Distanzen für das Kantengras separat gespeichert werden. Das Kantengras wird wie folgt aktualisiert, wenn die zurückgelegte Distanz in eine Richtung größer ist als der Kantengrasabstand $mEdgeSpacing$:

1. Graspatches von Graskanten löschen
   Mittels der Formel $\frac{mEdgeSpacing}{mGrassSpacing}$ wird errechnet, wie viele einzelne Patches aus der Reihe gelöscht werden müssen. Die Graskanten in Bewegungsrichtung müssen an beiden Enden (also die vordersten und hintersten Elemente der STL-Liste) gekürzt werden. Die von innen gezählte zweite Reihe wird so zur innersten (ersten) Reihe. Auch die direkten Nachbarn müssen angepasst werden: Sie müssen an dem Ende, das nicht in Richtung der Bewegung liegt, gekürzt werden. Die Eckpositionen des Kantengrases müssen ebenfalls aktualisiert werden, damit im nächsten Schritt die neuen Kanten generiert werden können.

2. Neue Kanten generieren
   Es müssen zwei komplett neue Kanten erstellt werden. Die eine wird ganz außen in Kamerarichtung generiert, die andere entsprechend auf der gegenüberliegenden Seite an der innersten Position, wo zuvor das Gras vorhanden war. Nicht zu vergessen ist nach diesem Schritt die erneute Aktualisierung der Eckpunkte des Kantengrases, da alle Positionsberechnungen relativ zu diesen geschehen.

3. Überflüssige Kanten löschen
   Die im ersten Schritt zum Teil gelöschte innerste Reihe wird nun komplett gelöscht. Ebenso entfällt die äußerste Reihe entgegen der Kamerabewegung. Damit ist gewährleistet, dass jede Seite wieder gleich viele Reihen besitzt.

4. Neue Graspatches anfügen
   Durch die vorherigen Schritte sind einige Lücken entstanden, die nun noch geschlossen werden müssen. Dies betrifft die seitlichen Reihen und die hintere Reihe. Die Prozedur ist ähnlich zum ersten Schritt, allerdings sind nun die anderen jeweiligen Enden der Kanten von einer Neugenerierung von Patches betroffen. Auch nach diesem Schritt müssen zwei Eckpositionen neu gesetzt werden (vgl. Abbildung 15).

Da Vertex Buffer Objects die mehrfach indirekte Datenstruktur (Strukturen in Listen innerhalb Listen) des Kantengrases, die sich zum Aktualisieren sehr gut eignet, nicht direkt nutzen können, müssen im Anschluss die Daten für die VBOs komplett neu erzeugt werden.

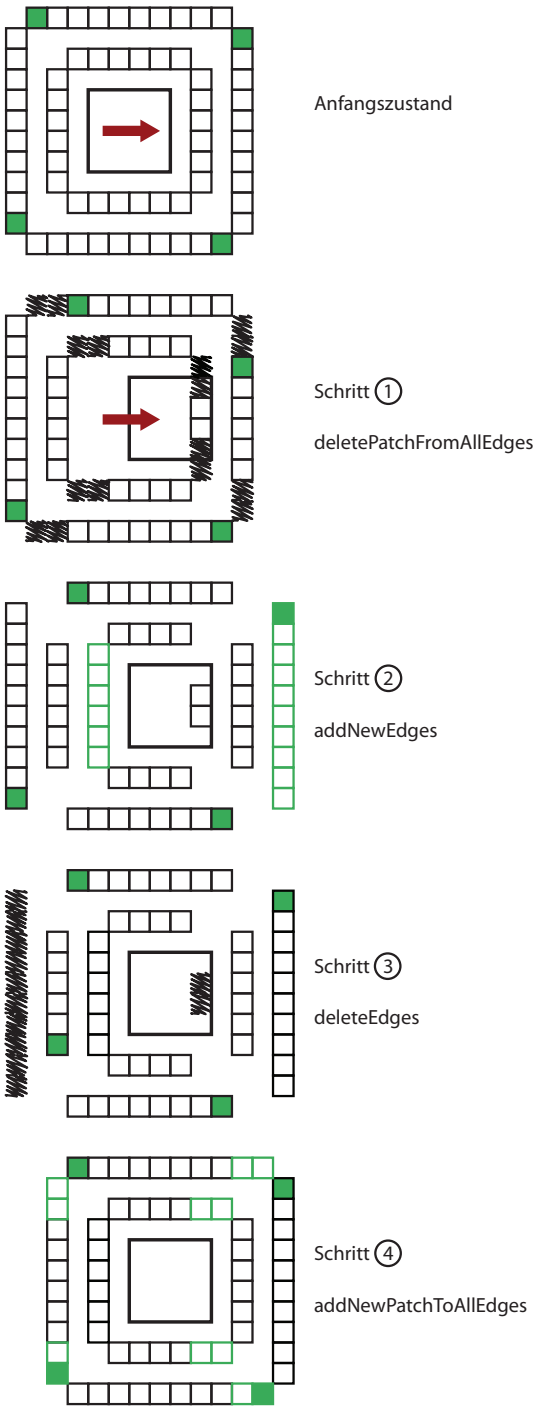

Abbildung 16: Aktualisierung des Kantengrases in Pfeilrichtung in vier Schritten. Im Zentrum befindet sich das Gras der feinsten Detailstufe. Die grünen ungefüllten Quadrate stellen die neu zu generierenden Patches dar, die grün gefüllten sind die Eckpositionen. Die schwarz schraffierten Patches müssen gelöscht werden.

8.6 Bodentextur und Terrain

Für den dritten Detaillevel wurde eine gekachelte Bodentextur mittels GIMP[22] erstellt. Die Problematik ist dabei die genaue Abstimmung auf das bereits vorhandene *echte* Gras. Ideal wäre hier eine direkt von der Simulation erstellte Textur. Leider verhindert der Ansatz der Texturquads einen einfachen Bildmitschnitt von oben auf das Gras als Textur, da eine solche Perspektive keinen seitlichen Blick auf die Texturquads zulässt und die Texturquads fast gänzlich verschwinden. Daher wurde die Bodentextur per Hand erstellt um die bestmögliche farbliche Übereinstimmung mit der zweiten Detailstufe zu erreichen. Sie wird mittels eines Shaders auf das Terrain gesetzt und gegebenenfalls bei Bergen in eine andere Textur übergeblendet. Verschiedene Terrains lassen sich über sogenannte Heightmaps laden. Diese Heightmaps sind Bilder, die Grauwerte im Bereich von 0 bis 255 enthalten. Ein Wert von 0 wird dabei auf eine flache Ebene abgebildet und ein Wert von 255 stellt die Spitze eines Berges dar. Durch Anpassung der y-Koordinate der Grasbüschel an die Terrainhöhe können so auch Wiesen auf hügeligem Gelände erstellt werden.

8.7 Bildqualitätsverbesserungen

Um die Darstellung weiter zu verbessern wurden zwei zusätzliche Techniken erprobt und implementiert. Die Beweggründe und Resultate werden im Folgenden vorgestellt.

8.7.1 Screen Space Ambient Occlusion

Die Verschattung zwischen den Gräsern geschieht bisher nur über die Grastextur, d. h. die Grastextur wird nach unten hin dunkler. Zusätzlich wird durch das Umknicken der Grasbüschel in Kapitel 9.2.6 eine Aufhellung der betroffenen Patches verursacht um das Umknicken zu betonen. Dies verursacht unter Umständen unschöne Aufhellungen an Stellen in der Nähe der Kollisionsobjekte, an denen normalerweise eine Verdunklung durch Kontaktschatten stattfinden. Mittels der Screen Space Ambient Occlusion (SSAO) Technik kann diese zusätzliche Verdunklung hinzugefügt werden. Zusätzlich ergänzt SSAO auch eine Verschattung zwischen allen Grasbüscheln.

SSAO ist ein bildraumbasiertes Verfahren, dass die ambiente Beleuchtung approximiert und erstmals von Crytek[23] für das Computerspiel Crysis verwendet wurde [Mit07]. Seitdem genießt es große Popularität in Spielen und wird vermehrt eingesetzt um die räumliche Wahrnehmung zu verbessern. Der durch das Verfahren errechnete Ambient-Occlusion-Wert

[22] http://www.gimp.org/
[23] http://www.crytek.com/

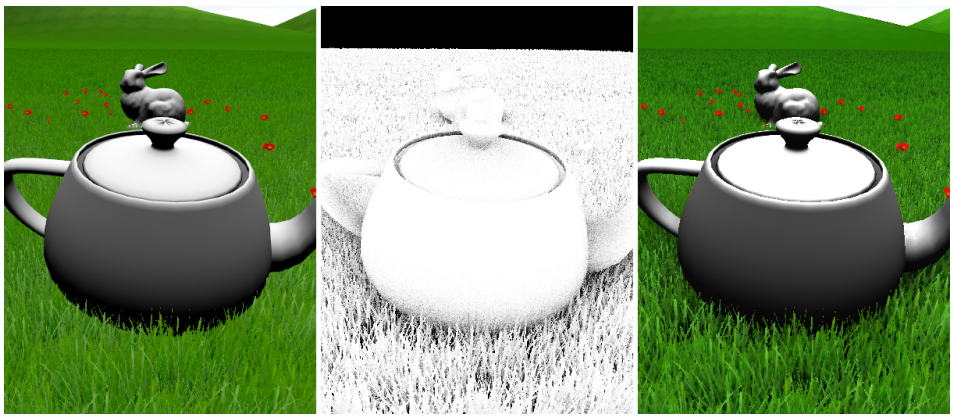

Abbildung 17: Screen Space Ambient Occlusion: Ohne SSAO (links), Ambient Occlusion Wert (Mitte), mit SSAO (rechts), jeweils mit diffuser Shader-Beleuchtung

zwischen 0 und 1 gibt an, wieviel ein zu beleuchtender Punkt von seiner Umgebung (Halbraum um diesen Punkt) sieht. Der Farbwert wird mit diesem SSAO-Wert gewichtet. Da das Verfahren im Bildraum (Screen Space) arbeitet, verwendet es für seine Berechnungen nur Daten, insbesondere Tiefenwerte, der im Bildraum sichtbaren Objekte.

Für jedes Pixel des Bildraums wird dessen Weltkoordinate bestimmt und von dieser Koordinate ausgehend in einem Radius bzw. Halbraum zufällige Samples generiert. Der *tatsächliche* Tiefenwert dieser Samples wird mit den Tiefenwerten aus einer Tiefentextur verglichen. Diese Tiefentextur beinhaltet die von der Kamera gesehenen *vordersten* Tiefenwerte. Für den Vergleich müssen die Samples in Weltkoordinaten in Bildschirmkoordinaten projiziert werden. Sollte der tatsächliche Tiefenwerte größer sein als derjenige aus der Textur, dann ist das Sample innerhalb von Geometrie gelandet und somit verdeckt. Der Ambient-Occlusion-Wert zwischen 0 und 1 errechnet sich dann aus dem Verhältnis zwischen den unverdeckten Samples und aller generierten Samples. Zusätzlich wird durch eine Gewichtungsfunktion verhindert, dass Samples, die sehr weit weg vom Ausgangspunkt landen (sehr großer Tiefenwert), einen großen Einfluss auf den AO-Term haben – naheliegende Samples werden dagegen stärker gewichtet. Das Ergebnis dieser Berechnung ist ein bildraumgroßes Bild, das Grauwerte für jedes Pixel zwischen 0 und 1 enthält. Für die letztendliche Nutzung wird dieses verrauschte Ergebnis unter Berücksichtigung der Kanten im Bild noch verwischt (kantenerhaltender Blur-Filter).

Der in dieser Diplomarbeit verwendete SSAO-Shader wurde erstmals für die Vorlesung Echtzeitrendering entwickelt und für die Grassimulation angepasst. Da semi-transparente Texturen verwendet werden, muss für diese der Shader modifiziert werden. Im Shader, der die für den SSAO-

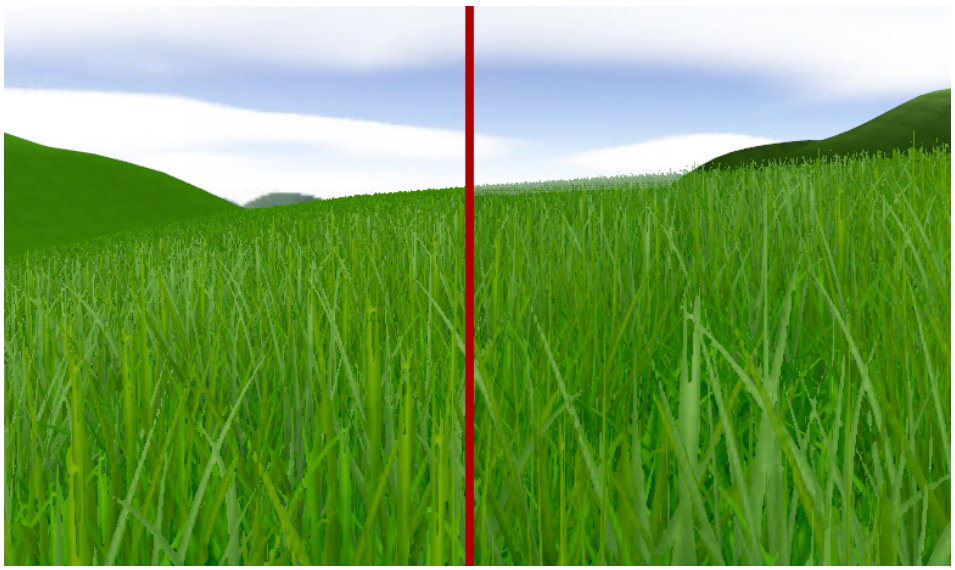

Abbildung 18: Alpha To Coverage: an (links) und aus (rechts). Dithering wird am Grashorizont sichtbar.

Shader benötigten Daten wie z. B. Positionen und Normalen in verschiedene Texturen gleichzeitig rendert[24], muss zusätzlich der Alphawert des aktuellen Pixels gesichert werden. Dieser ergibt sich im Falle des Grases aus dem Alphawert der Textur. Dadurch wird gewährleistet, dass der SSAO-Shader die eigentlich darzustellende Form des Grasbüschels berücksichtigt anstatt die rechteckige tatsächliche Form des Texturquads. Für die Berechnung des AO-Wertes werden in dieser Simulation nur die Graspatches und die Kollisionsobjekte berücksichtigt, indem nur deren Daten mit dem MRT-Shader in Texturen gespeichert werden. Die Skybox für die Darstellung des Himmels und das Terrain werden nicht berücksichtigt, da diese zu falschen AO-Werten und zusätzlichem Rauschen bei der Berechnung führen würden. Dadurch wird allerdings auch keine Verdeckungsberechnung zwischen Gras und Terrain durchgeführt, was aber bei einem visuellen Vergleichstest kaum erkennbar war, da die Grasbüschel eng genug beieinander stehen und damit genug Verdeckungsmöglichkeiten bereit stehen. Abbildung 17 zeigt das Ergebnis des Screen Space Ambient Occlusion Verfahrens.

8.7.2 Alpha To Coverage

Weiterhin besteht auf Grund der Nutzung der semi-transparenten Grastexturen ein Aliasing Problem. Bei Nutzung des Alphatests für Texturen wird eine Stelle innerhalb der Textur entweder angezeigt oder gar nicht angezeigt. Werden diese Texturen dann auf einem Quad vergrößert angezeigt,

[24] Multiple Render Targets (MRT)

entstehen durch das bilineare Filtern innerhalb des Alpha-Kanals der Textur unschöne Ausfransungen an den Gräserrändern [Gre07].

Alpha To Coverage (siehe OpenGL Reference Pages [Gra10]) ist eine auf Multisampling basierende Technik um auch innerhalb der Texturen Anti-Aliasing anwenden zu können. Multisample-Anti-Aliasing (MSAA) verhindert die Treppchenbildung nur bei tatsächlicher Geometrie, nicht aber bei Texturen. MSAA muss aktiv sein um Alpha To Coverage verwenden zu können. Wenn MSAA über `glEnable(GL_MULTISAMPLE)` aktiviert wurde, wird für jedes Fragment (welches durch Geometrie generiert wird) eine Überdeckungsmaske (Wert zwischen 0.0 und 1.0) generiert, die aussagt, wie sehr das Pixel von diesem Fragment verdeckt ist. Ein Wert von 1.0 bedeutet, dass das Pixel komplett vom Fragment überdeckt ist, 0.0 entsprechend keine Verdeckung, ansonsten eine teilweise Überdeckung. Wird nun Alpha To Coverage zusätzlich mit `glEnable(GL_SAMPLE_ALPHA_TO_COVERAGE)` aktiviert, wird der Alpha-Wert des Fragments mit der Überdeckungsmaske verUNDet. Das Ergebnis wird gedithert und über mehrere Multisample Samples verteilt [Per05]. Abbildung 18 zeigt die gleiche Szene mit und ohne Alpha To Coverage. Das Dithering ist bei flachem Betrachtungswinkel an den Grasspitzen und bei naher Betrachterperspektive erkennbar. In den Vordergrund rückt durch aktiviertes Alpha To Coverage aber das ansonsten weichere Erscheinungsbild der Gräser.

Da GLUT nicht mehr weiterentwickelt wird und auf dem Stand des Jahres 2001 ist, verwendet der durch GLUT erstellte OpenGL-Kontext die alte Multisample Extension `GLX_SAMPLE_SGIS`. Diese wird zudem unter Microsoft Windows nicht unterstützt. Für funktionierendes Alpha To Coverage werden die OpenGL Extensions `GL_MULTISAMPLE_ARB` und `GL_SAMPLE_ALPHA_TO_COVERAGE_ARB` benötigt. Das GLFW Toolkit unterstützt den für diese Extension benötigten OpenGL-Kontext. Um einen solchen Kontext zu erstellen, genügt das Aufrufen der Funktion `glfwOpenWindowHint` mit dem Parameter `GLFW_FSAA_SAMPLES` und als Wert die Anzahl der gewünschten Samples. Diese Funktion setzt das Pixelformat des zu erzeugenden Fensters auf ein Format, das Multisampling ermöglicht. Nun müssen beide Extensions wie oben erwähnt nur noch aktiviert werden um sie zu benutzen.

9 Animation

Aufbauend auf den theoretischen Grundlagen aus Kapitel II kommt eine erweiterte und angepasste Version der Cyclon Physik-Engine zum Einsatz. Die in [Mil07] entwickelte Physik-Engine funktioniert entweder nur als Partikel-Physik-Engine oder als Starrkörper-Physik-Engine. Für die Grassimulation werden allerdings sowohl Partikel als auch Starrkörper benötigt. Daten-

typen und Funktionen müssen entsprechend hinzugefügt oder angepasst werden.

Die zentrale Klasse der Physik-Engine ist die `PhysicsSystem` Klasse, die stark erweitert wurde. Zugriff auf die Physik-Engine ist nur durch Methoden dieser Klasse gegeben. Zunächst wird nun die Initialisierung und der allgemeine Ablauf erklärt, ehe in den darauffolgenden Kapiteln näher auf die einzelnen Bestandteile der Physiksimulation eingegangen wird.

9.1 Initialisierung des Physiksystems

Zunächst wird anhand der Konfigurationsdatei (siehe Kapitel 11.1) festgelegt, ob das Partikelsystem eine Interpolation verwenden soll (siehe Kapitel 9.3) und ob es zum Teil mittels GPU (siehe Kapitel 9.4) berechnet wird. Sind diese Variablen gesetzt worden, geht es an die eigentliche Initialisierung der Komponenten des Physiksystems.

Partikel

Falls die Interpolation verwendet wird, werden $\frac{numberGrassPatchesX^2}{4}$ Partikel erzeugt (bzw. bei ungerader Anzahl an Graspatches entsprechend $\frac{(numberGrassPatchesX+1)^2}{4}$), andernfalls $numberGrassPatchesX^2$. Die Partikel werden an den Stellen der Graspatches generiert (siehe Abbildung 21 links). Ihre y-Koordinate entspricht der Terrainhöhe an dieser Stelle plus die Constraintlänge, die vorher festgelegt wird. Jeder Constraint wird mit seinem Ankerpunkt, der Länge und dem dazugehörigen Partikel gespeichert. Für die Partikel werden außerdem die physikalischen Werte Geschwindigkeit $(0.2, 0.2, 0.2)$, Beschleunigung $(0, 1, 0)$, Masse 1 und eine geringe Dämpfung 0.97 gesetzt. Letzterer Wert stellte sich bei einem visuellen Vergleichstest unterschiedlicher Werte als der Realität am nächsten kommend heraus. Die konstante Beschleunigung in Richtung $(0, 1, 0)$ führt dazu, dass das Partikel immer gerade nach oben strebt und damit bei einer Kollision wieder zum Ausgangspunkt zurückkehrt.

Falls die GPU zur Berechnung des Partikelsystems herangezogen wird, werden zusätzlich die hierfür benötigten Daten, Texturen, Shader etc. initialisiert. Eine genaue Auflistung der zusätzlichen Datenstrukturen befindet sich in Kapitel 9.4.

Wind

Auch die in der Simulation vorherrschenden Winde müssen initialisiert werden. Der `ParticleWinds` Klasse wird eine Referenz auf die Partikel und die Anzahl der Grasreihen übergeben, sowie eine globale Windgeschwindigkeit ($windSpeed$). Die Windgeschwindigkeit wird beeinflussen, wie schnell die Grasreihen später nacheinander vom Wind betroffen sind. Ein hoher

Wert bedeutet, dass die Grasreihen fast zeitgleich vom Wind umgebogen werden, ein niedriger lässt den Wind langsam über die Wiese ziehen und die Grasreihen sichtbar nacheinander umknicken. Nun müssen die einzelnen Winde definiert werden. Man kann bestimmen, in welche Richtung (positive oder negative z-Richtung) der Wind weht und wie sehr die Gräser in dieser Richtung umgebogen werden. Außerdem muss in Sekunden angegeben werden, wie lange dieser Wind anhält. In die Grassimulation wurde folgendes Windschema integriert, damit sich das Gras hin und her wiegt:

1. Wind in z-Richtung $(0, 1, 0.9)$, 5 Sekunden lang
2. Kein Wind $(0, 0, 0)$, 6 Sekunden lang (dadurch gelangt das Gras wieder in seine Ausgangssituation)
3. Wind in negativer z-Richtung $(0, 1, -0.9)$, 4 Sekunden lang
4. Kein Wind, 6 Sekunden

Auch andere Windmuster oder Zeitangaben wären problemlos möglich. Das Windschema wird nach einem Durchlauf der vier Winde mit dem ersten Wind wieder von vorne wiederholt.

Starrkörper

Zusätzlich zu den Partikeln und Winden werden exemplarisch drei Kollisionsobjekte (Starrkörper, in der Simulation von der Klasse `RigidBody`) erstellt, davon zwei Objekte mit einem Kugelkollisionsprimitiv und ein Objekt mit einem Würfel als Kollisionsprimitiv.

Für die beiden Kugeln müssen folgende Angaben gemacht werden:

- Radius, Masse, Massetensor (nach Formel 11)
- Geschwindigkeit, Beschleunigung, Position, Orientierung
- Dämpfung (wird benötigt um Energie, die durch numerische Instabilitäten des Integrators enstehen, zu entfernen)
- Einschlaffähigkeit (nach bestimmten Kriterien wird der Körper nicht länger aktiv berechnet und ruht dann)
- Modell und Textur für das darzustellende Objekt (Fußball bzw. Stanford Bunny[25])

Für den Würfel muss anstelle des Radius die halbe Kantenlänge und der Massetensor nach Formel 12 angegeben werden. Als Modell dient eine einfache texturierte Box.

[25] http://graphics.stanford.edu/software/scanview/models/bunny.html

9.2 Ablauf pro Frame

In jedem Frame wird die Methode `runPhysics` mit dem Übergabeparameter `duration` aufgerufen um das Physiksystem zu aktualisieren. Die `duration` ist dabei die Zeit, die die Anwendung für alle Berechnungen zwischen zwei Frames benötigt. Sie kann mit Hilfe der Funktion `glfwGetTime` berechnet werden, indem der letzte gemessene Zeitpunkt (jeweils immer vor Aufruf von `glfwSwapBuffers`) von der aktuellen Zeit abgezogen wird. Auch eine von der Renderinggeschwindigkeit unabhängige Zeit wäre möglich.

Innerhalb der `runPhysics` werden folgende Schritte erledigt:

1. Windkräfte anwenden
2. Kollisionen mit Starrkörpern erkennen
3. Die im vorherigen Schritt erfassten Kollisionen behandeln
4. Bei Benutzung der GPU: Daten für die GPU vorbereiten bzw. aktualisieren
5. Integration der Partikel und Starrkörper
6. Kontakte auf Grund der Partikelconstraints generieren und auflösen
7. Die Graspatches in ihrer Neigung aktualisieren

9.2.1 Windkräfte

Zunächst wird die Windsimulation berechnet. Neben den zuvor definierten einzelnen Winden müssen in zwei STL-Vektoren laufend die Startzeit, wann der aktuelle Wind angefangen hat zu wehen, und der zur Zeit aktive Wind pro Grasreihe gespeichert werden. Zu Anfang wird für alle Grasreihen der aktive Wind auf den ersten definierten Wind gesetzt. Die Startzeiten ergeben sich bei Programmstart reihenweise nach der Formel $startTime = elapsedTime + \frac{row}{windSpeed}$. Die Reihen ($row$) werden dabei von 0 bis $numberParticlesZ$ durchlaufen.

Die Winde werden reihenweise (also in $+z$ oder $-z$ Richtung) auf das Gras angewendet. Ob der Wind anzuwenden ist, wird über die $elapsedTime$ pro Reihe berechnet:

$$\begin{aligned} &(elapsedTime \;<\; startTime + activeDuration) \quad \&\& \\ &(elapsedTime \;>\; startTime) \end{aligned}$$

Die $activeDuration$ bezieht sich dabei auf die zuvor angegebene Zeitspanne, die der jeweilige aktuelle Wind andauert. Je nach Windrichtung werden dann die Reihen entweder vorwärts oder rückwärts durchlaufen und die

Windkraft $direction_{wind}$ auf die bereits wirkenden Kräfte der Partikel dieser Reihen hinzugefügt. Ein Wind darf keine Kraft auf die Partikel ausüben, solange dessen $startTime$ noch nicht erreicht ist, also falls $elapsedTime < startTime$ gilt.

Wenn die Zeitspanne vorüber ist, muss der nächste Wind (im Falle des letzten Winds wird mit dem ersten erneut begonnen) als aktiver Wind gesetzt werden und die $startTime$ für die Grasreihe auf die aktuelle $elapsedTime$ gesetzt werden. Die zur Initialisierung versetzt gesetzten Startzeiten können so beibehalten werden. Das Ergebnis ist ein simulierter Wind, der je nach Windrichtung zunächst die nächstgelegenen Grasbüschel zur Seite kippt, ehe dessen Nachbarn beeinflusst werden. Da auch der echte Wind eine Richtungskomponente besitzt, entspricht dieses zeitversetzte Verhalten näherungsweise der Realität.

9.2.2 Kollisionserkennung

Auf die Windkräfte folgt die Kollisionserkennung zwischen Partikeln und Starrkörpern sowie zwischen den Starrkörpern selbst. Die Kollisionserkennung wird wie in Kapitel 6 beschrieben in zwei Phasen, der groben und der genauen, unterteilt.

Die grobe Erkennung zwischen Partikeln und Starrkörpern wurde auf Grund der gleichmäßig verteilten Partikel mittels der Gitterstruktur (siehe Kapitel 6.1) realisiert. Es werden nur diejenigen Partikel betrachtet, die im Gitter gesehen in der Nähe der Starrkörper liegen und damit potenzielle Kollisionskandidaten sind. Auf diese Partikel kann direkt mittels Index zugegriffen werden, weshalb außer der Berechnung des Start- und Endindexes keine zusätzlichen aufwändigen Berechnungen auf Partikeln ausgeführt werden müssen, die sowieso nicht mit Starrkörpern kollidieren. Abbildung 19 zeigt die benötigten Indizes für die Kugel und den Würfel. Ausgehend von deren Körpermittelpunkten bzw. -positionen kann mittels Radius bzw. Umkreis einfach eine Weltkoordinate bestimmt werden, die von oben gesehen links oben bzw. rechts unten um den Körper liegt. Diese Koordinate wird wiederum auf den benötigten Anfangs- und Endindex umgerechnet. Diese Indizes sind zwar sehr großzügig gewählt, können dafür aber ohne großen Aufwand berechnet werden und enthalten garantiert alle möglichen von einer Kollision betroffenen Partikel. Diese im STL-Vektor zwischen Anfangs- und Endindex liegenden Partikel werden in der genauen Kollisionserkennung untersucht und mögliche Kollisionsdaten werden generiert. Da das Hauptaugenmerk dieser Simulation auf der Kollisionserkennung zwischen Partikeln und Starrkörper liegt, werden die drei Starrkörper der Einfachheit halber paarweise gegeneinander getestet. Für sie liegt keine grobe Kollisionserkennung vor.

Die genaue Kollisionserkennung prüft nun, ob sich die möglichen Partikel bzw. Starrkörper tatsächlich berühren. Jedes dieser Partikel (bzw. jeder

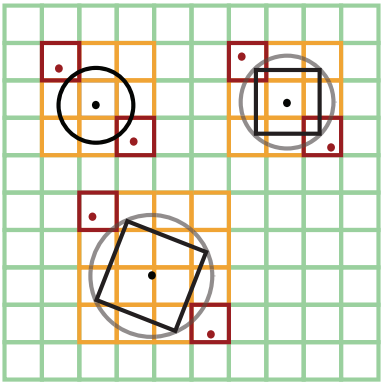

Abbildung 19: Grobe Kollisionserkennung von Starrkörpern (schwarz). Rote Punkte sind die Eckweltkoordinaten, die roten Quadrate die dazugehörigen Graspatches (bzw. deren Indizes). Alle orangenen Patches sind potenzielle Kollisionskandidaten.

dieser Starrkörper) wird gegen alle übrigen potenziellen Partikel bzw. Starrkörper getestet. In einer Kontaktdatenstruktur werden der Kontakt, die Kollisionsnormale und die Eindringtiefe gespeichert. Ihre Berechnungen erfolgen nach den Formeln aus Kapitel 6.2.

9.2.3 Kollisionsbehandlung

Die Kollisionen aus dem vorherigen Schritt müssen nun behandelt und aufgelöst werden (siehe Abbildungen 20 und 27). Hier wird zu großen Teilen auf die Implementation der Cyclon Physik-Engine zurückgegriffen (z. B. für die Kollisionsbehandlung zwischen den Starrkörpern). Sie verläuft nach den in Kapitel 7 erläuterten Grundlagen und Formeln. Sie muss also die neuen Positionen und Geschwindigkeiten der Partikel bzw. Körper berechnen und anwenden. Da die Cyclon Physik-Engine allerdings keine Starrkörper mit Partikeln zusammen simulieren kann, wurden entsprechend Berechnungen speziell für die Partikel ergänzt. Da Partikel keine winkelabhängigen Größen besitzen, können die neuen Positionen der Partikel allein durch die Eindringtiefe und die Kontaktnormale ermittelt werden. Die neue Position berechnet sich aus

$$newPos = oldPos + contactNormal \cdot \left(-\frac{penetration}{totalInertia}\right)$$

wobei $totalInertia$ die komplette aufsummierte Trägheit der an dieser Kollision beteiligten Objekte ist. Dadurch wird erreicht, dass die Partikel eine stärkere Positionsänderung vollziehen anstelle des eher trägen Starrkörpers.

Auf eine Geschwindigkeitsänderung für die Partikel wurde verzichtet. Die neue Geschwindigkeit ist somit identisch mit der alten. Auf Grund der

Abbildung 20: Kollisionsbehandlung von einem Starrkörper, der über die Wiese bewegt wurde

konstanten Beschleunigung in Richtung $(0, 1, 0)$ und des Partikelconstraints streben die Partikel stets nach oben zu ihrer Ausgangsposition.

9.2.4 Partikel- und Starrkörperintegration

Neben der Änderung der Geschwindigkeiten, Positionen, etc. durch eine Kollision müssen die Daten der Partikel und Starrkörper fortwährend durch Integration aktualisiert werden. Obwohl dies verglichen zur Kollisionsbehandlung recht wenige Berechnungen sind (siehe Kapitel 5.1.2 für Partikel und 5.2.2 für Starrkörper), müssen sie immer auf alle Partikel und auf alle Starrkörper angewendet werden. Speziell für die Starrkörper kann eine zusätzliche Variable festgelegt werden, die aussagt, ob der Körper gerade ruht oder aktiv ist. Wenn der Körper ruht, wird dieser bis zur nächsten Kollision nicht mehr integriert. Für die in dieser Simulation verwendeten Partikel ist eine solche Einschlaffähigkeit nicht vorgesehen, da der Wind permanent vorhanden ist und damit eine dauerhafte Integration aller Partikel nötig macht.

9.2.5 Partikelconstraint

Die Partikel werden nach Kapitel 7.2.1 einfach linear zurückprojiziert:

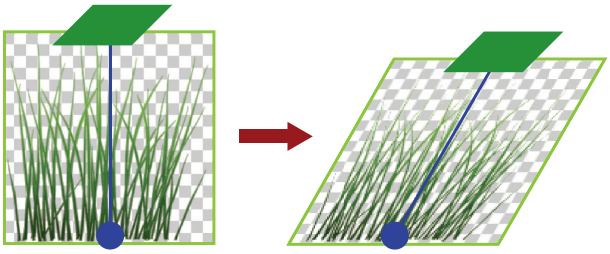

Abbildung 21: Aktualisierung des Grases bei Neigung entsprechend des Partikels

$$\begin{aligned} \vec{c} &= normalise(\overrightarrow{anchor} - \vec{p}_{particle}) \\ \vec{p}^{new}_{particle} &= \overrightarrow{anchor} - \begin{pmatrix} constraintDistance \cdot c_x \\ constraintDistance \cdot c_y \\ constraintDistance \cdot c_z \end{pmatrix} \end{aligned}$$

Die Anker der Constraints und die $constraintDistance$ wurden zuvor in einer Kontaktdatenstruktur (wie bei der Kollisionserkennung) erstellt. Damit ist das Partikelconstraint gelöst.

9.2.6 Abschließende Aktualisierung des Grases

Nachdem die Partikel nun berechnet wurden, muss die Darstellung der Gräser angepasst werden. Die Anpassung der Schräglage aller Graspatches durch Aktualisierung der Vertex Buffer Objects ist somit der letzte Schritt der `runPhysics` Methode.

Anhand der Partikelpositionen werden die Texturquads angepasst (siehe Abbildung 21). Entscheidend sind die oberen zwei Vertices des Quads, deren x-, y- und z-Koordinaten wie folgt geändert werden:

$$\begin{pmatrix} -cos_{Rotation} + posX_{particle} - mScaleXZ \\ terrainheight + mScaleY - smaller \\ +sin_{Rotation} + posZ_{particle} - mScaleXZ \end{pmatrix} \quad Links\ oben$$

$$\begin{pmatrix} +cos_{Rotation} + posX_{particle} - mScaleXZ \\ terrainheight + mScaleY - smaller \\ -sin_{Rotation} + posZ_{particle} - mScaleXZ \end{pmatrix} \quad Rechts\ oben$$

Der Parameter $smaller$ berechnet sich mit $constraintDistance - (posY_{particle} - terrainheight)$ und wird maximal $\frac{constraintDistance}{2}$ groß. Damit wird sicher gestellt, dass das Grasquad nicht zu kurz dargestellt wird und somit die einzelnen Texturquads deutlich sichtbar werden. Das Quad und somit auch die Textur werden durch dieses Verfahren verzerrt. Durch

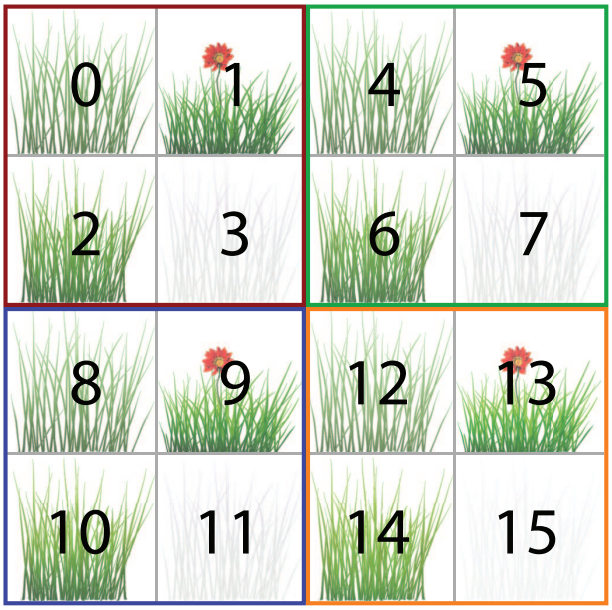

Abbildung 22: Texturatlas mit Indexnummern (von normal nach hell: rot, grün, blau, orange)

das Partikelconstraint wird verhindert, dass die Verzerrung zu groß und damit erkennbar wird.

Neben der Verschiebung der Vertices wird das Umknicken der Grashalme auch durch eine Texturänderung hervorgehoben. Wann die Textur geändert wird, hängt von der Neigungshöhe $tiltingHeight = posY_{particle} - posY_{constraintanchor}$ ab. Ein wenig Neigung führt zu einer etwas helleren Textur:

$$constraintDistance - 0.3 \leq tiltingHeight < constraintDistance - 0.2$$

bzw.

$$constraintDistance - 0.4 \leq tiltingHeight < constraintDistance - 0.3$$

Die größte Neigung wird am hellsten dargestellt:

$$tiltingHeight < constraintDistance - 0.4$$

Die Änderung der Textur wird mittels Addieren eines Indexoffsets ($+4$, $+8$ und $+12$) vollzogen. Der Texturatlas ist hierzu wie in Abbildung 22 aufgebaut.

9.3 Interpolation

Bisher wurde für jedes Grasquad ein eigenes Partikel verwendet, um seine Bewegung zu beeinflussen. Um die Performance zu verbessern, wurde eine

Interpolation eingebaut, d. h. es wird nur noch für jedes vierte Graspatch ein Partikel erzeugt. Folgende Interpolationsfälle (siehe auch Abbildung 23) entstehen dadurch bei der Aktualisierung der Grasbüschel:

- Keine Interpolation: Das Partikel befindet sich genau an dem betrachteten Graspatch.
- Interpolation zwischen zwei Partikeln: Das zu aktualisierende Patch befindet sich zwischen zwei Gräsern.
- Interpolation zwischen vier Partikeln: Das Patch befindet sich genau in der Mitte vierer Partikel.

Bei einer ungeraden Anzahl an Graspatches in x- bzw. z-Richtung müssen spezielle Randfälle beachtet werden, die aber ebenfalls maximal eine Interpolation zwischen zwei Partikeln notwendig machen.

Sämtliche Graspatches werden in einer einfachen Schleife von $k = 0$ bis Anzahl Patches durchlaufen um sie anhand der Partikel (und Constraints) zu aktualisieren. Für die Interpolation werden folgende Zahlen benötigt, da bei aktiver Interpolation die Anzahl der Partikel ungleich der Anzahl der Patches ist und daher der Index $k_{particle}$ für die Partikel (und Constraints) erst berechnet werden muss:

$$k_{interpolated} = \frac{k}{numberGrassPatchesX}$$

$$k_{interpolated}^{remainder} = k \,\%\, numberGrassPatchesX$$

Keine Interpolation

Falls keine Interpolation nötig ist, da das Graspatch direkt über ein Partikel verfügt, so kann dies mit der Abfrage

$$k_{interpolated} \,\%\, 2 \neq 0 \quad \&\& \quad k_{interpolated}^{remainder} \,\%\, 2 \neq 0$$

überprüft werden (wobei % für die modulo-Operation steht). Das Partikel (bzw. Constraint) befindet sich in seinem STL-Vektor dann an der Stelle

$$k_{particle} = \frac{k_{interpolated}}{2} \cdot (numberParticlesX) + \frac{k_{interpolated}^{remainder}}{2}$$

Daraus ergibt sich die benötigte Position für das Grasbüschel:

$$\vec{p}_{interpolated} = particleAt(k_{particle})$$

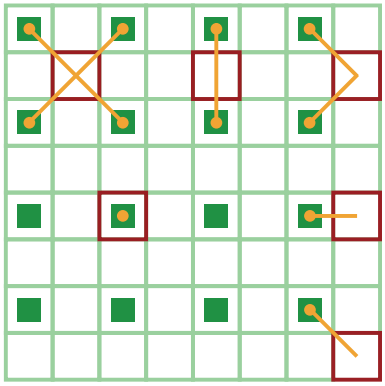

Abbildung 23: Interpolationsfälle für die roten Graspatches: Keine Interpolation, Interpolation zwischen vier oder zwei Partikeln und Randfälle

„Zweier"-Interpolation

Als Beispiel für eine tatsächliche Interpolation sei die Berechnung der Interpolation zwischen zwei Partikeln (Normalfall, in Abbildung 23 oben Mitte) wie folgt aufgeführt: Ein solcher Fall kann mit der Abfrage

$$k_{interpolated} \,\%\, 2 \neq 0 \quad \&\& \quad k^{remainder}_{interpolated} \,\%\, 2 = 0$$

erkannt werden. Das obere Partikel befindet sich in seinem STL-Vektor an der Position

$$k_{particletop} = \frac{k_{interpolated}}{2} \cdot (numberParticlesX) + \frac{k^{remainder}_{interpolated}}{2}$$

und das untere an der Stelle

$$k_{particlebottom} = k_{particletop} + numberParticlesX$$

Die interpolierte Position (analog der interpolierte Constraint) für dieses Graspatch ergibt sich dann aus

$$\vec{p}_{interpolated} = \frac{particleAt(k_{particletop}) + particleAt(k_{particlebottom})}{2}$$

Die weiteren Fälle werden ähnlich durch Divisionen und Modulo-Rechnungen ermittelt.

9.4 Berechnungen auf der GPU

Alle bisherigen Berechnungen wurden ausschließlich von der CPU berechnet. Für die Integration der Partikel kommt auch die Grafikkarte (GPU) in Frage. Die Verwendung der Grafikkarte für allgemeine Aufgaben wie z. B.

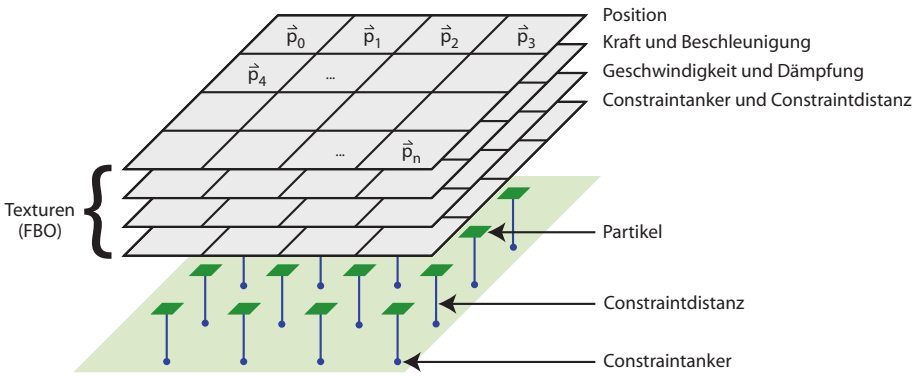

Abbildung 24: Analogie zwischen Partikeln und Texturen und Aufbau der Texturen zur GPU-Berechnung

solche Physikberechnungen wird auch als General Purpose Computation on Graphics Processing Unit, kurz GPGPU, bezeichnet. Die Grafikkarte zeichnet sich durch ihre hohe parallele Verarbeitungsgeschwindigkeit aus. Die Partikelintegration muss auf tausende Partikel angewendet werden und für jedes Partikel muss die gleiche Berechnung durchgeführt. Da diese Berechnungen unabhängig voneinander sind, bietet sich die GPU hierfür an. Auch die Partikelconstraints können zeitgleich auf der GPU berechnet werden.

Die Daten der Partikel und der Constraints müssen hierzu an die Grafikkarte übergeben werden. Dies geschieht mittels Texturen. Für die Integration und Behandlung der Constraints werden vier Texturen `GL_TEXTURE_2D` mit dem Datenytyp `GL_FLOAT` benötigt:

- Partikelpositionen (r,g,b)
- Geschwindigkeit (r,g,b) und Dämpfung (a)
- Kraft (r,g,b) und Beschleunigung (a)
- Constraintanker (r,g,b) und Constraintdistanz (a)

Jedes Texel der Textur enthält korrespondierend zum Partikel und dessen Constraint die jeweiligen Daten. Diese Entsprechung ist der Grund, wieso die Partikel in eben dieser Reihenfolge im STL-Vektor dauerhaft gehalten werden – dadurch ist eine weitere Korrespondenzdatenstruktur unnötig. Abbildung 24 veranschaulicht diesen Zusammenhang. Zusätzlich wird die *duration* als `uniform float` an den Shader übergeben. Falls die in Kapitel 9.3 vorgeschlagene Interpolation verwendet wird, sind entsprechend auch die Texturen kleiner.

Die eigentlichen Berechnungen können für die GPU 1:1 übernommen werden und werden mittels eines Fragment-Shaders erledigt. An ein Frame Buffer Object werden zwei Ausgabetexturen gebunden, die die neuen

Positionen und Geschwindigkeiten enthalten. Die oben bereits erstellten Texturen mit den Positionen bzw. Geschwindigkeiten dürfen nicht wiederverwendet werden, da in einer Textur nicht gleichzeitig gelesen und geschrieben werden darf. Mittels der Multiple Render Targets Technik kann in die beiden Ausgabetexturen gleichzeitig geschrieben werden. Die Ergebnisse müssen für die weitere Physiksimulation (z. B. für die Starrkörper) bekannt sein. Daher müssen die Texturen ausgelesen und in einen STL-Vektor für die CPU gespeichert werden. Hierfür stehen die Befehle `glReadBuffer` und `glReadPixels` zur Verfügung (siehe auch Kapitel 13.1).

10 Grafische Benutzeroberfläche

Für die grafische Benutzeroberfläche (Graphical User Interface, GUI) wird wie eingangs erwähnt die Bibliothek AntTweakBar verwendet. Die GUI wird direkt mit OpenGL gerendert und erlaubt das einfache Anpassen von Parametern. Buttons ermöglichen es Methoden im Programmcode aufzurufen.

10.1 Wiese und Terrain ändern

Die Grassimulation kann ganz erheblich mittels zweier Buttons geändert werden. Es ist möglich unterschiedliche Heightmaps zu laden. Nach Eingabe des Dateinamens einer Bitmap-Bilddatei und dem Betätigen des Ladebuttons wird das Terrain anhand der Höhendaten dieser Datei neu erstellt. Sowohl das Gras als auch das Kantengras werden anhand der veränderten Höhendaten neu gesetzt und befinden sich nach erfolgreichem Laden der Heightmap wieder auf dem Terrain.

Die zweite Möglichkeit, die Grassimulation maßgeblich zu ändern, besteht in der Änderbarkeit der Gras- und Kantengrasparameter und der damit verbundenen Neugenerierung der kompletten Wiese. Änderbar sind die Parameter $mEdgeSpacing$ (als ein Vielfaches vom fest vorgegebenen $mGrassSpacing$) und die Anzahl der Reihen für das Kantengras, sowie die Anzahl der Reihen für das Gras (gilt sowohl für die x- als auch für die z-Richtung). Nach der Neugenerierung wird die Wiese komplett gelöscht und neu berechnet. Dies ist nicht in Echtzeit möglich und dauert je nach eingegebenen Parametern unterschiedlich lange. Je größer die Anzahl der Reihen gewählt wird, desto länger dauert die Berechnung aller Positionen und sonstiger Daten.

10.2 Interaktive Parametereinstellungen

Neben diesen zwei Eingriffen in die Grassimulation, die aufwändige Neuberechnungen zur Folge haben, gibt es viele weitere Einstellungsmöglichkeiten, um die Simulation neben der Kameraführung und dem Bewegen

der Kollisionsobjekte mittels Maus und Tastatur interaktiv zu beeinflussen. Diese Funktionalitäten teilen sich auf in die Kategorien Kamera, Physik, Gras und (sonstiges) Visuelles.

Kamera

- `First Person Mode`: Die Kamera hält automatisch einen gewissen Abstand zum Terrain ein. Dadurch kann man sich in der Simulation wie in einem Spiel in der Ersten-Person-Perspektive bewegen.
- `Grass follows camera`: Durch diese Option lässt sich das Level-of-Detail-System deaktivieren. Die Wiese wird sich dann nicht mehr mit der Kamera mitbewegen.

Physik

- `Velocity movement`: Falls aktiviert, werden die Objekte mit den Tasten T, F, G, H, U und J mittels einer Geschwindigkeitsänderung bewegt anstelle einer einfachen direkten Positionsänderung.
- `Select and control body`: Hier kann das zu bewegende Objekt (Fußball/Mower, Bunny oder Box) ausgewählt werden.
- `Show physics`: Dies ist hauptsächlich eine Debuggingfunktion, mit der die Physiksimulation (Kollisionsprimitive, Partikel und Constraints) illustriert wird.

Grass

- `Scale Grass`: Hier kann die y- und xz-Skalierung verändert werden. Diese Option ändert die Höhe bzw. Breite der Graspatches (Gras und Kantengras gleichermaßen).

Visuelles

- `Use Alpha To Coverage`: Falls MSAA aktiv ist, kann zusätzlich Alpha To Coverage (siehe Kapitel 8.7.2) für die Graspatches aktiviert werden.
- `SSAO`: Durch das Aktivieren des ersten Unterpunktes wird auf das Bild das SSAO-Verfahren (siehe Kapitel 8.7.1) unter Berücksichtigung der Farbinformationen und eines Blurfilters angewendet und eine einfache Beleuchtung mittels eines Fragment-Shaders erzeugt, da in diesem Fall die Standard OpenGL Beleuchtung nicht funktioniert.

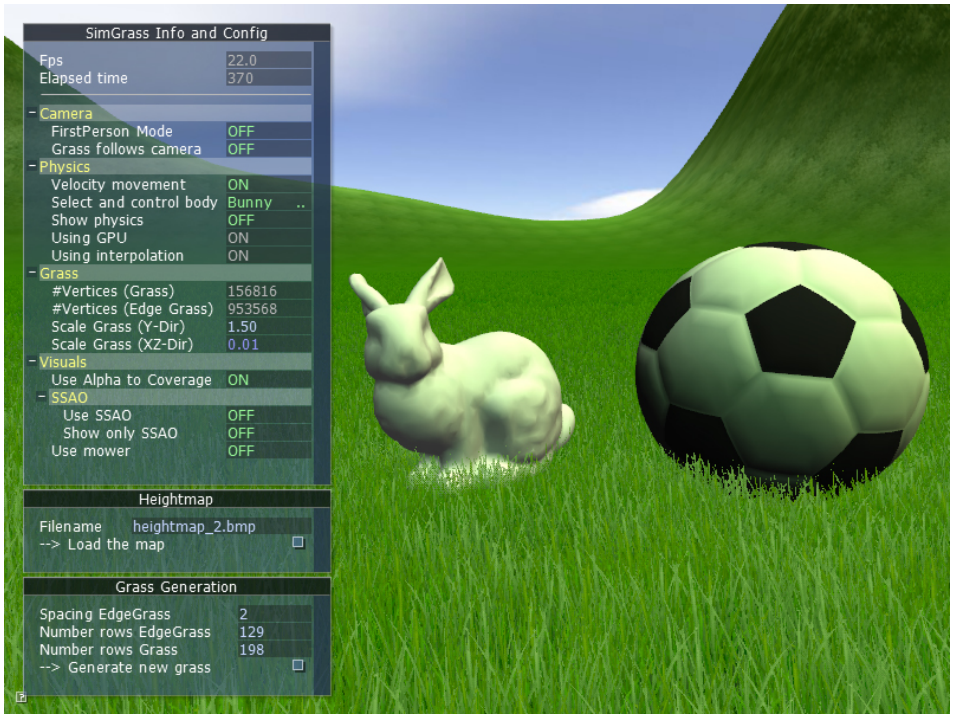

Abbildung 25: Screenshot der Grassimulation mit GUI

- `Use mower`: An die Stelle des Fußballs tritt ein Rasenmäher, der nach der Selektion entsprechend über die Wiese bewegt werden kann (siehe auch Kapitel 11.3).

Abbildung 25 zeigt das komplette Interface.

11 Zusätzliche Features

Um weitere Einstellungen zu erlauben und die Einsatz- und Erweiterungsmöglichkeiten der Grassimulation zu demonstrieren, wurden die folgenden Features implementiert.

11.1 Konfigurationsdatei

Einige wenige Optionen können und sollen nicht zur Laufzeit geändert werden. Diese Parameter sind:

- `MSAA_active`: Aktiviert oder deaktiviert das Multisample-Anti-Aliasing (wird für Alpha To Coverage benötigt). Da MSAA an den OpenGL-Kontext gebunden ist, müsste beim Umschalten das Fenster neu erstellt werden. Daher kann dieser Wert nur vorab definiert werden (1 an oder 0 aus).

- `MSAA_samples`: Gibt die Anzahl der verwendeten Samples für das MSAA an. Die maximale Anzahl hängt von der jeweiligen Grafikkarte ab. Aktuelle Grafikkarten können bis zu 16 Samples[26] verwenden. Je höher dieser Wert ist, desto langsamer wird die Darstellung, aber desto besser werden die Kanten geglättet.
- `physics_GPU`: Bei Aktivierung wird die GPU für Teile der Physikberechnung (siehe Kapitel 9.4) herangezogen.
- `physics_interpolation`: Wird die Interpolation auf 1 (an) gesetzt, so werden nur noch $\frac{1}{4}$ der üblichen Partikel verwendet (siehe Kapitel 9.3).

11.2 Fußballfeld

Mit diesem Feature soll der prinzipiell mögliche Variantenreichtum bei der Darstellung des Grases hervorgehoben werden. Durch die Nutzung eines Texturatlasses ist es ein Leichtes eine Grastextur zu integrieren, die ein weiß angemaltes Grasbüschel darstellt. Beim Generieren der Grasdaten, speziell des Indexes innerhalb des Atlasses, an einer Position (i, j) muss zusätzlich abgefragt werden, ob an dieser Position eine bestimmte Textur verwendet werden soll. Im Falle der Fußballplatzmarkierung geschieht dies über die Funktion `hasFootballMarking`, die einen bool-Wert zurückliefert. Die Fußballfeldmarkierung wird in dieser Implementation bei Benutzung des Level-of-Detail-Systems allerdings entfernt. Hier wäre also Potenzial weitere Verbesserungen einzubringen (siehe auch Kapitel 15).

11.3 Rasenmäher

Die Nutzung von Texturquads ermöglicht die einfache Implementation eines Rasenmähers mit dem spielerisch die Skalierungsmethode für die Texturen betrachtet werden kann. Die Kollisionserkennung wird wie üblich verwendet. Sobald die Grasbüschel anhand der Positionen der Partikel in ihrer Schräglage aktualisiert werden, greift der Rasenmäher: Ist dieser aktiv, wird der Texturindex des betroffenen Graspatches mit dem einer helleren Variante im Texturatlas dauerhaft ersetzt (vgl. Kapitel 9.2.6). Bei der Erstellung der VBOs wird der $scaleYMinus$ Wert von der y-Koordinate ebenfalls dauerhaft subtrahiert.

[26] Der OpenGL Extensions Viewer http://www.realtech-vr.com/glview/ zeigt beispielsweise die maximale Sampleanzahl an.

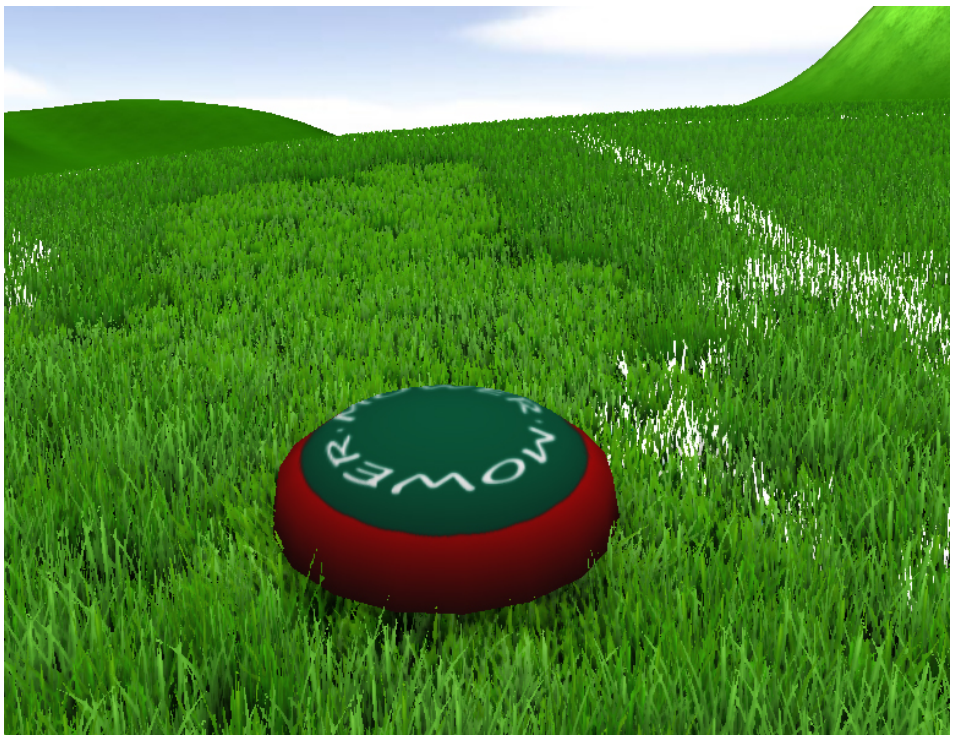

Abbildung 26: Rasenmäher und teilweise gemähtes Gras (mit aktiviertem SSAO)

Abbildung 27: Gras erholt sich von einer Kollision mit einer Kiste

Teil IV
Ergebnisse und Bewertungen

Abschließend wird die erzielte Grassimulation bewertet. Sowohl die Qualität der Darstellung und der Animation als auch die erzielte Performance werden dabei untersucht. Neben den Vorteilen dieser Grassimulation werden auch die Nachteile kritisch beurteilt und mögliche Lösungs- und Verbesserungsansätze diskutiert. Abschließend wird ein Blick auf die Zukunft von Grassimulationen geworfen.

12 Darstellungs- und Animationsqualität

Um die Visualisierung und die Animation möglichst objektiv zu bewerten, wird die Grassimulation zunächst mit anderen Verfahren verglichen, ehe die Realität als Referenz herangezogen wird.

12.1 Vergleich zu bisherigen Verfahren

Für einen Vergleich können nur solche Veröffentlichungen zum Einsatz kommen, die entweder eine ausführbare Demo oder Videos zur Verfügung stellen, um einen umfangreicheren Eindruck zu erhalten. Für jede Renderingmethode wird repräsentativ ein Verfahren herangezogen:

- Bildbasiertes Verfahren: Rendering Countless Blades of Waving Grass [Pel07]
- Volumenbasiertes Verfahren: Making Grass and Fur Move [Ban06]
- Level-of-Detail-System (inkl. geometrischer Repräsentation): Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting [Bou08]

Diese Verfahren erlauben nur einen Vergleich der Darstellung, da sie keine Kollisionsbehandlung unterstützen. Die Animationsqualität wird daher mit folgenden Verfahren verglichen:

- Real-Time Continuum Grass [CJ10]
- A Procedural Approach to Animate Interactive Natural Sceneries [GPR+03]

Der Vergleich findet mit Best-Case-Szenarien der implementierten Grassimulation statt, da anzunehmen ist, dass insbesondere bei Videos ebenfalls die besten Szenen zu sehen sind. Auf die Worst-Case-Szenarien, also Probleme und Nachteile der Grassimulation, wird in Kapitel 14 vertieft eingegangen.

Abbildung 28: Reduzierung der Gräser bei großer Entfernung vom Betrachter und Alphablending in *Rendering Countless Blades of Waving Grass* [Pel07]

12.1.1 Darstellungsqualität

Die Darstellungsqualität hängt bei bildbasierten Verfahren maßgeblich von der eingesetzten semi-transparenten Textur ab. Im Fall von [Pel07] erscheint die Textur etwas zu gelblich und teilweise sind Ränder an den Grasbüscheln zu erkennen (siehe Abbildung 28). Dies ist dem Alphablending zu verschulden, kann aber normalerweise durch eine gute Textur bzw. einen guten Alphawert (siehe auch Abbildung 3) verhindert werden. Die in dieser Diplomarbeit erstellte Grassimulation nutzt statt des aufwändigeren Alphablendings nur den Alphatest. Die Kanten des Grases sind daher in [Pel07] weicher. Pelzer verwendet zwar kein Level-of-Detail-Verfahren, reduziert stattdessen aber je nach Entfernung zum Gras die Anzahl der Billboards und blendet sie (von transparent zu undurchsichtig) dann langsam wieder ein. Dies ist sehr deutlich in Abbildung 28 erkennbar. Die Bodentextur ist jedoch so gewählt, dass diese Überblendung deutlich sichtbar wird. Eine Anpassung der Farbe hat anders als in *SimGrass* nicht stattgefunden.

Das volumenbasierte Verfahren bietet eine sehr realistische Darstellung aus der Vogelperspektive. Dafür sind die einzelnen Shells aus einem flachen Winkel zu erkennen. *SimGrass* ist für die Vogelperspektive nicht ausgelegt (siehe Kapitel 4), weshalb die Texturquads aus einer solchen Perspektive deutlich zum Vorschein kommen (siehe Abbildung 29). Erst bei flachen Betrachtungswinkeln wirkt das Gras voluminös. Die Simulationen verhalten sich also genau entgegengesetzt.

Das Level-of-Detail-System von [Bou08] arbeitet einwandfrei und lässt die Übergänge zwischen den Detailstufen kaum erahnen. Die Darstellung ist sehr detailliert, ebenso die Beleuchtung inkl. Schattenwurf. Durch die geo-

Abbildung 29: Vogelperspektive (1. und 3. Bild) und First-Person-Perspektive (2. und 4. Bild) im Vergleich: *SimGrass* (links) und [Ban06] (rechts)

metrische Repräsentation des Grases in Betrachternähe kann die Bildqualität überzeugen. Auf Grund des Level-of-Detail Systems sind viele Grashalme darstellbar, ähnlich wie bei *SimGrass*. Insbesondere bei hügeligen Landschaften stößt allerdings das in *SimGrass* eingesetzte Level-of-Detail-System an seine Grenzen (siehe Abbildung 38).

Zusammenfassend lässt sich sagen, dass die Bildqualität entscheidend von den eingesetzten Texturen abhängt. Geometriebasierte Verfahren können für den Nahbereich eine detailliertere Darstellung bieten. Alphablending sieht insbesondere bei sehr naher Betrachtung einer semi-transparenten Textur weicher aus als bloßes Alphatesting, benötigt aber eine sortierte Wiese zur Anwendung. Das in dieser Arbeit eingesetzte Alpha To Coverage kann die gezackten Kanten einer Textur bei aktiviertem Alphatest nur geringfügig verbessern. Mögliche bessere Verfahren werden in Kapitel 15 vorgestellt.

12.1.2 Animationsqualität

Die bisher verglichenen Veröffentlichungen verwenden entweder gar keine Animation ([Bou08]) oder implementieren nur Wind bzw. Gravitation ([Ban06] und [Pel07]). Obwohl in der Veröffentlichung des Billboard-Ansatzes nur die Bewegung der zwei oberen Vertices eines Grasquads beschrieben wird, bewegt sich das gesamte Grasquad in der zur Verfügung gestellten interaktiven Demo. Die insgesamte Bewegung ist zudem sehr zittrig und entspricht kaum einer Bewegung durch Wind. Der Wind aus [Ban06] verhält sich auf der gesamten Wiese wie ein punktueller Wirbel und daher sehr chaotisch. Eine Windböhe, die konstant über das Gras weht wie bei *SimGrass*, wird nicht simuliert.

Auch diejenigen Verfahren, die Kollisionen mit Objekten simulieren können, besitzen eine Windsimulation. Chen et al. [CJ10] kann Wind als

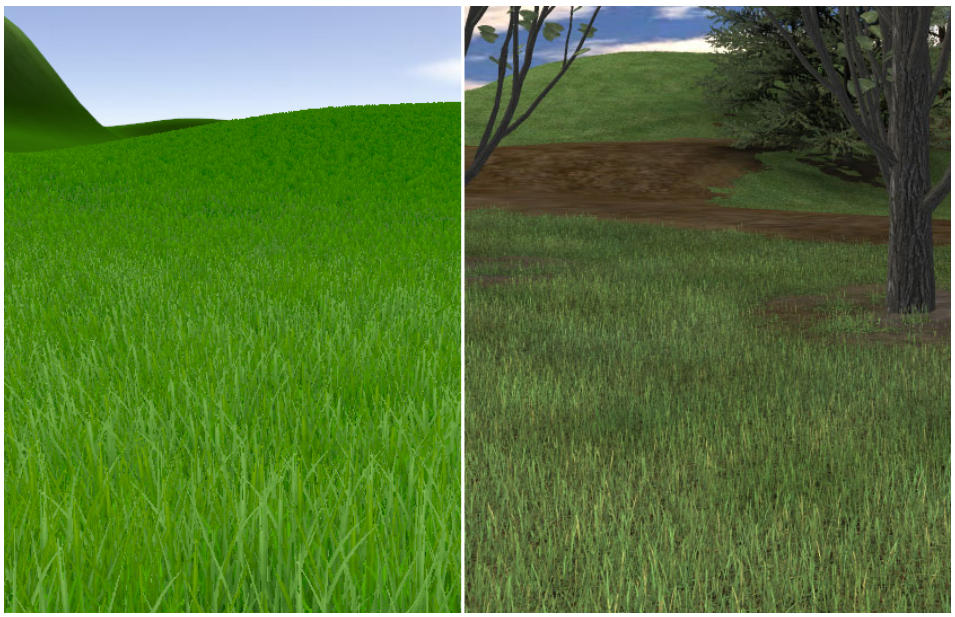

Abbildung 30: Vergleich des Level-of-Detail-Systems mit [Bou08]

Effekt auf die gesamte Wiese simulieren, aber auch Wind von z. B einem Helikopter ausgehend. Ihre Windsimulation ist umfangreich und individuell anpassbar. Die Windsimulation von *SimGrass* ist in der Windrichtung nur eingeschränkt anpassbar und beherrscht keine komplexeren Windeffekte. Der Effekt eines Helikopters für eine festgelegte Flughöhe ließe sich mit Hilfe eines Kugelkollisionsprimitiv annähern, wobei das tatsächlich darstellbare Objekt höher positioniert wird als das eigentliche Primitiv. In [GPR+03] ist Wind ebenfalls ohne bestimmte Windrichtung (aber mit verschiedenen Charakteristika wie z. B die eines Windstoßes oder Tornados) und nur durch zeitversetztes Schwenken aller Gräser implementiert. Die meisten Windsimulationen bewegen also die Grasbüschel willkürlich und unabhängig voneinander. Für zwei Windrichtungen ist die Windsimulation von *SimGrass* an eine zeitliche Komponente gebunden und die Gräser bewegen sich nacheinander in Richtung des Winds, ehe sie wieder in ihre Ruheposition zurückkehren.

Kollisionsbehandlung

Der Ansatz von Chen et al. [CJ10] beinhaltet die Kollisionsbehandlung von kurzem und langem Gras mit mehreren konvexen und konkaven Objekten. Die Neigung des Grases wird an umliegende Gräser weiter propagiert, was einer wellenartigen Ausbreitung gleich kommt. Wie schnell und flächig sich die Ausdehnung verhält, kann eingestellt werden. Bei großer Ausdehnung sieht die Grassimulation unnatürlich aus, da einzelne Grashalme in der

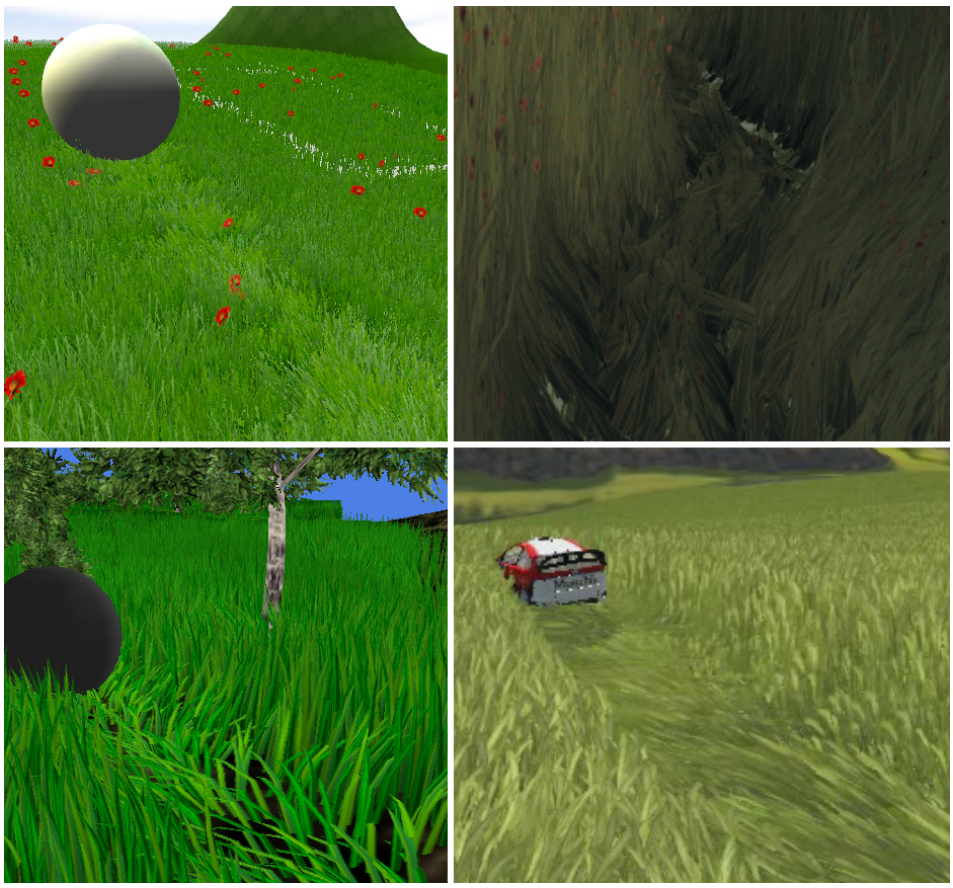

Abbildung 31: Kollisionsbehandlungen im Vergleich (von links oben im Uhrzeigersinn: *SimGrass*, [ORSK09], [CJ10], [GPR+03])

Realität sehr dünn sind und kaum benachbarte Gräser beeinflussen. Eine Propagation ist in *SimGrass* daher nicht vorhanden. Bisher sind zudem nur einzeln einsetzbare Kugel- und Würfelprimitive implementiert, die somit keine konkaven Objekte darstellen können. Hier wäre allerdings eine Erweiterung durch Kombination mehrerer Primitive denkbar. Insgesamt gesehen verhalten sich beide Kollisionssimulationen sehr unterschiedlich, ermöglichen aber beide auf großen Wiesen die Kollision mit unterschiedlichen Objekten.

Guerraz et al. [GPR+03] verwenden ein Trittprimitiv zur Kollision mit Gras. Einzelne Grashalme knicken bei Berührung einer Kugel beispielsweise merklich weg und nach Entfernung wieder zurück. Da sie bei der feinsten Detailstufe eine geometrische Repräsentation verwenden, ist diese Animation sehr genau. In *SimGrass* wird hingegen nur das Texturquad als ganzes verzerrt bzw. verkürzt ohne einen wirklichen Knick darzustellen.

Orthmanns *GPU-based responsive grass* [ORSK09] kann leider nur an

Abbildung 32: Vergleich mit der Realität: mit SSAO (links), ohne SSAO (mittig), Referenz (rechts)

Hand von Bildern grob bewertet werden. Durch die Nutzung von Billboards ist bei niedergedrücktem Gras ähnlich wie bei *SimGrass* die Quadstruktur erkennbar. Eine genaue Spur im Gras durch Kollisionsobjekte ist sichtbar. Als Kollisionsobjekt kommt ein Modell eines Menschen zum Einsatz, das aus kugelähnlichen Primitiven besteht. Detailliertere Objekte sind jedoch auch möglich. Die Grasbewegungen auf Grund einer Kollision sind durch Unterteilung des Billboards genauer – ob dies einen visuellen Vorteil bringt, ist anhand der Bilder nicht erkennbar. Die Gräser scheinen sehr tief eindrückbar zu sein. Für *SimGrass* erschien es zweckmäßig, die Tiefe zu begrenzen, da andernfalls die einzelnen Quads zu stark sichtbar werden würden. Abbildung 31 zeigt die verschiedenen Abdrücke durch die Kollisionsverfahren der Publikationen im Gras.

12.2 Vergleich mit der Realität

Als Referenz für jede Grassimulation muss die Realität dienen. In dieser Diplomarbeit stützen sich die Beobachtungen zu großen Teilen auf einen ca. 8 cm kurzen Zierrasen. Für die Darstellungsqualität ist die verwendete Textur, Texturvielfalt und Beleuchtung entscheidend. Mit einer passenden Textur kann mit einer einfachen Standard-OpenGL-Beleuchtung bereits eine gute Darstellung gelingen, die der Realität nahe kommt (siehe Abbildung 32). Auffällig ist allerdings die fehlende Varianz in der Beleuchtung und die farbliche Monotonie. Durch weitere Texturen könnte mehr Abwechslung erzeugt werden. Verschattungen in der Realität lassen sich aber wohl schwer mit dem Billboardansatz abbilden, da einzelne Grashalme nicht berücksichtigt werden können.

Kurzes Gras bewegt sich auf Grund von Wind nur sehr wenig und kaum

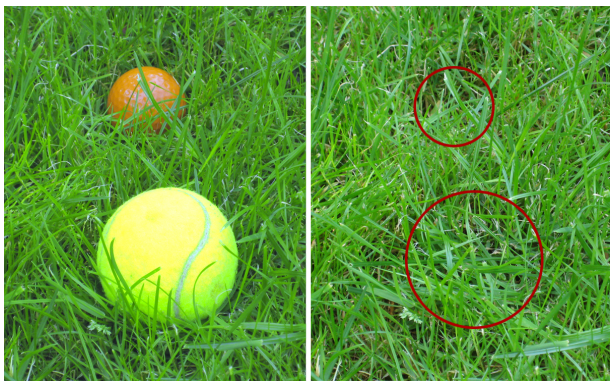

Abbildung 33: Kollision in der Realität: mit Objekten (links) und kurz nach Entfernung der Objekte (rechts, rot markiert die Lage der Objekte)

sichtbar, weshalb die in der Simulation erstellten Winde ebenfalls nur eine geringe Auswirkung auf das Gras haben.

Für die Kollisionserkennung von Gegenständen mit (zumindest kurzem) Gras hat sich gezeigt, dass es zu keiner wellenartigen Ausbreitung der Neigung kommt, sondern das Phänomen regional auf das Objekt begrenzt ist (siehe Abbildung 33). Auswirkungen sind nur direkt unter dem Objekt auszumachen, wobei sich einige wenige Grashalme um das Objekt herum bewegen, da das Gras mitunter etwas verflochten ist. Auf Grund dieser Beobachtung wurde auf eine Gras-Gras Kollision verzichtet und der Fokus auf die eigentliche Kollision gelegt. Doch auch diese ist bei kurzem Gras nur schwer zu erkennen. Von weitem ist nur noch die Aufhellung der zerdrückten Grasstellen zu beobachten. Eine solche findet in der Nähe des Betrachters auch in *SimGrass* statt. Da aus weiter Entfernung von einer Kollision kaum mehr etwas zu erkennen ist, kann wie im Level-of-Detail-System durchaus darauf verzichtet werden. Da kurzes Gras oft zerzaust ist, verhält sich die Regeneration zur Ursprungsposition ohne erkennbares Muster, da sich einzelne Grashalme durch die Kollision ineinander verheddern. Eine Grasspur ist nur sehr kurz zu erkennen und die Gräser erholen sich recht schnell. Diese Details werden mit der Grassimulation allerdings nicht dargestellt, da dazu die einzelnen Grashalme hätten simuliert werden müssen. Die Partikel kehren vor allem bei einer Würfelkollision meistens von einer Richtung aus zurück an ihre Ruheposition, was sehr unnatürlich wirkt.

13 Performance

Im Folgenden wird die Performance bestimmter Programmteile der Simulation bewertet. Die implementierten Beschleunigungsmöglichkeiten, namentlich die Nutzung einer Interpolation und der GPU zur Physik-Berechnung können aktiviert und deaktiviert werden und die entsprechenden Ergebnis-

se verglichen werden. Zusätzlich wird die Performance des Level-of-Detail-Systems und der groben Kollisionserkennung gemessen. Im Anschluss wird die Gesamtperformance bewertet und der Einfluss unterschiedlicher Einstellungen der Wiese auf die Performance betrachtet.

Als Testsystem steht folgender Computer zur Verfügung:

Desktop-PC mit einem Intel Core2Duo E6600 (2,4 Ghz), 2 GB Arbeitsspeicher, Nvidia Geforce GTX 260-216 mit 896 MB Grafikspeicher, Windows 7 x64

13.1 GPU-Berechnung

Die GPU wird für das Berechnen der Integration aller Partikel und deren Constraints eingesetzt. Das Zurückschreiben der Daten für die CPU ist allerdings sehr langsam und stellt den Flaschenhals bei der gesamten GPU-Berechnung dar. Je mehr Partikel berechnet werden müssen, desto größer werden auch die Texturen, in denen die Partikeldaten gespeichert werden müssen und die an die GPU übergeben werden müssen. Zum Vergleich wurde die Zeit gemessen, die für die Integration der Partikel benötigt wurde. Im Falle der GPU wurde das Hochladen und Runterladen der Daten mitgemessen. Abbildung 34 (links) zeigt den Unterschied in der Ausführungsgeschwindigkeit in Millisekunden. Solange die Partikelanzahl und somit die Texturgröße noch recht klein ist, ist auch die GPU Implementation noch recht schnell. Die Berechnungsdauer steigt danach allerdings stärker exponentiell an als die der CPU-Implementation. Bei 250.000 Partikeln ist die CPU zwar doppelt so schnell wie die GPU, bei beiden Ansätzen können allerdings keine akzeptablen Frameraten mehr erreicht werden.

Betrachtet man nur die Ausführungsgeschwindigkeit des Fragment-Shaders (siehe Abbildung 34 rechts), der die Integration übernimmt, so ist zu erkennen, dass die GPU sehr effizient arbeitet und schneller als die CPU rechnet. Allein die Aufrufe `glReadPixels` und `glReadBuffer` machen ca. $\frac{1}{3}$ der Rechendauer der gesamten GPU-Implementation aus. Hinzugezählt werden müssen insbesondere noch die Rückführung der Daten in die Datenstrukturen der Partikel und das erneute Hochladen der Texturen an die GPU für die nächste Integration.

Aus diesen Diagrammen wird ersichtlich, dass die GPU zwar sehr wohl geeignet ist, die unabhängigen Berechnungen der Partikel auszuführen, es allerdings an der Übergabe der Daten an die GPU (und zurück) scheitert. Diese Übergabe ist derart langsam, dass die CPU insgesamt schneller rechnet.

13.2 Interpolation

Die Interpolation soll dazu dienen weniger Partikelintegrationen und Constraints lösen zu müssen. Dies hat zwei Konsequenzen: Es müssen einerseits

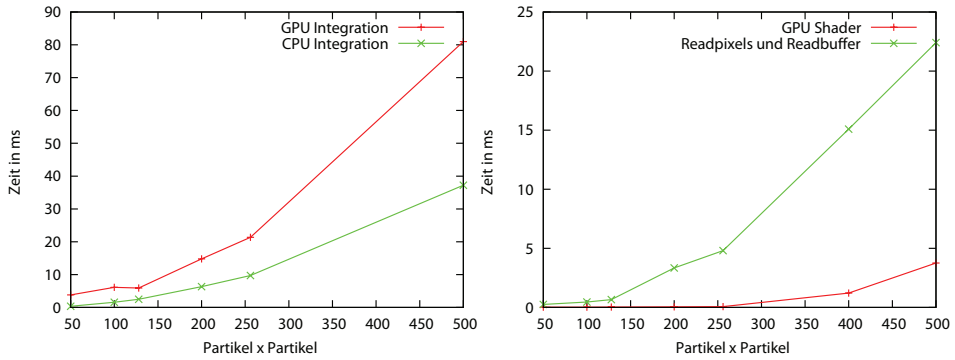

Abbildung 34: Links: Zeitmessung für die GPU und die CPU Integration. Rechts: Ausführungsgeschwindigkeit des Fragment-Shaders für die Integration und Geschwindigkeit der `Readpixels`- und `Readbuffer`-Operationen (auf zwei Texturen)

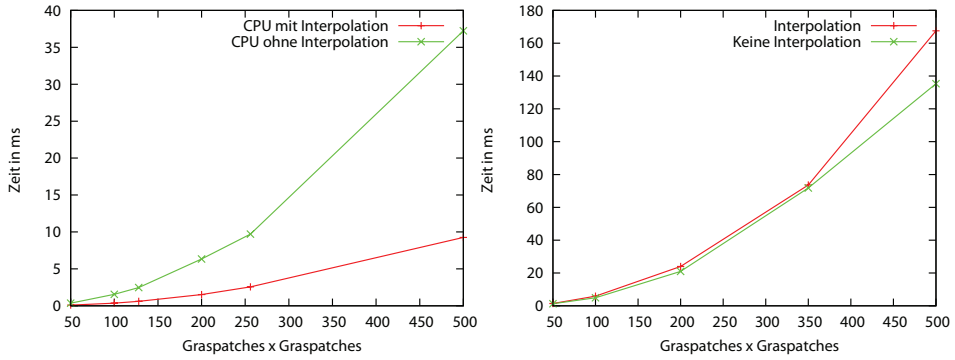

Abbildung 35: Links: CPU-Integration mit und ohne Interpolation. Rechts: Update der Grasneigungen mit und ohne Interpolation.

tatsächlich weniger Partikel bzw. Constraints berechnet werden, andererseits müssen im Anschluss die Ergebnisse für Graspatches, die keine Partikel besitzen, interpoliert werden. Abbildung 35 (links) zeigt, dass der Vorteil der Integration steigt, umso mehr Partikel verwendet werden, da damit die Anzahl der eingesparten Rechnungen ebenfalls steigt. Die Integration mit Interpolation ist 4-mal schneller als ohne, da $\frac{3}{4}$ der Partikel eingespart werden können. Wie die Abbildung 35 (rechts) zeigt, hält sich der Mehraufwand zum Aktualisieren der Graspatchneigungen bedingt durch die Interpolation in Grenzen. Bis 122.500 Partikeln ist kaum ein Unterschied zu erkennen, erst danach steigt der Mehraufwand etwas an. Zusammenfassend kann man sagen, dass die Interpolation einen deutlichen Performancegewinn bringt. Da visuell kein Unterschied in der Darstellung zu erkennen ist, sollte sie auf jeden Fall verwendet werden.

13.3 Grobe Kollisionserkennung

Für das Grasfeld wurde eine gitterbasierte, grobe Kollisionserkennung implementiert. Für drei Objekte in der Szene wurde getestet, inwiefern die Nutzung einer groben Kollisionserkennung einen Performancegewinn darstellt. Für die Zeitmessung wurden über 22.800 Partikel erstellt und all diese zunächst direkt mit den drei Kollisionsobjekten auf Kollision getestet. Dies dauerte 2,89 Millisekunden. Durch Hinzufügen der groben Kollisionserkennung, also nur dem Testen der zu den Objekten naheliegenden Partikel mit den Objekten, ergab sich eine Berechnungsdauer von nur 0,08 Millisekunden. Die Nutzung einer groben Kollisionserkennung wirkt sich also bereits bei wenigen Partikeln sehr positiv (hier z. B. 35-mal schneller) aus.

13.4 Level-of-Detail-System

Im Zusammenhang mit dem Level-of-Detail-System ist es interessant zu sehen, wie die Performance bei Bewegung der Kamera auf Grund der Anzahl der Kantengrasreihen beeinflusst wird. Die Zeiten wurden mit deaktiviertem Physiksystem (aber mit dem Update der Partikelpositionen) und bei Bewegung in Richtung der Ecken des Grasfeldes gemessen, wobei die Zeit jeweils über die Dauer eines kompletten Frames gemessen wird. Zu beachten ist, dass die Anzahl der Graspatches nicht linear mit der Anzahl der Kantengrasreihen steigt, da die äußeren Reihen mehr Graspatches beinhalten als die inneren. Dennoch zeigt sich, dass das Hinzufügen von 30 Grasreihen immer gleich viel Performance kostet, egal wieviele Grasreihen bereits existieren (siehe Abbildung 36: gleicher Anstieg der Zeit von 30 auf 60 und von 60 auf 90 Reihen). Eine Verdopplung der Kantengrasreihen führt ebenso nicht zu einer Verdopplung der Rechenzeit. Das Hinzufügen neuer Kantengrasreihen, das Löschen einzelner Graspatches usw. geschieht somit unabhängig von der Anzahl der Reihen bzw. Anzahl der Graspatches innerhalb einer Reihe. Anders sieht es bei der Erhöhung der Partikel bzw. der Gräser der feinsten Detailstufe aus. Diese müssen bei Bewegung der Kamera komplett kopiert werden, um innerhalb der STL-Vektoren die Datenreihenfolge beibehalten zu können. Der Performanceverlust ist daher bei steigender Partikelanzahl deutlich erkennbar.

13.5 Gesamtperformance

Die Gesamtperformance (siehe Abbildung 37) hängt maßgeblich von der Bewegung des Betrachters und von der Anzahl der Graspatches (Partikel) ab. Sie wird ohne und mit Bewegung des Betrachters (für die Performance des Level-of-Detail-Systems siehe auch Kapitel 13.4) mit unterschiedlichen Parametern gemessen.

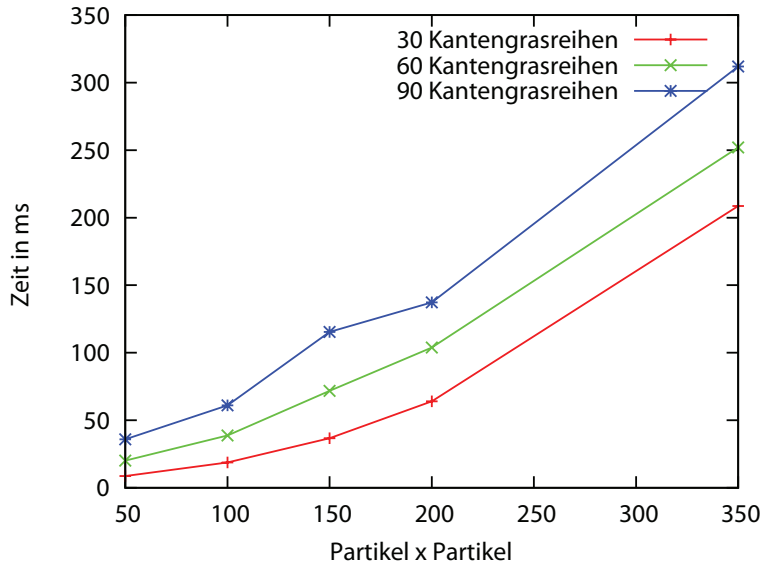

Abbildung 36: Zeitmessung für die gesamte Applikation (ohne Physiksystem) bei Bewegung des Betrachters in Richtung einer Ecke des Grasfelds

Bei Generierung von jeweils 30 Kantengrasreihen und ca. 22.800 Partikeln kann die Simulation ohne Bewegung des Betrachters mit fast 60 fps angezeigt werden. Die Aktivierung von MSAA hat kaum Einfluss auf die Performance, bietet aber wegen der Glättung aller Kanten (im Zusammenhang mit Alpha To Coverage auch Kanten innerhalb der Texturen) eine insgesamt bessere Bildqualität. Das Hinzufügen von über fünfmal mehr Kantengrasreihen drückt die Performance vergleichsweise wenig, da diese nicht animiert werden. Anders sieht es bei Steigerung der Partikelanzahl aus: 2500 Partikel mehr bedeuten einen Performanceverlust von ca. $\frac{1}{3}$ der Frames pro Sekunde.

Bei Bewegung über das Gras zeigt sich, dass die Performance stark einbricht. Dies ist den vielen Kopieroperationen und dem Neuaufbau der Vertex Buffer Objects geschuldet.

Ingsesamt zeigt sich dennoch, dass das Ziel einer Echtzeitsimulation mit Verbesserungspotenzial der Performance nach oben erreicht wurde. Für die Integration dieser Simulation in eine bestehende Anwendung müsste vor allem das Level-of-Detail-System verbessert werden (siehe Kapitel 15).

14 Vor- und Nachteile der Grassimulation

Texturen

Die Verwendung von semi-transparenten Texturen auf Quads erwies sich als eine schnelle Möglichkeit um viel Gras darzustellen. Grasbüschel können

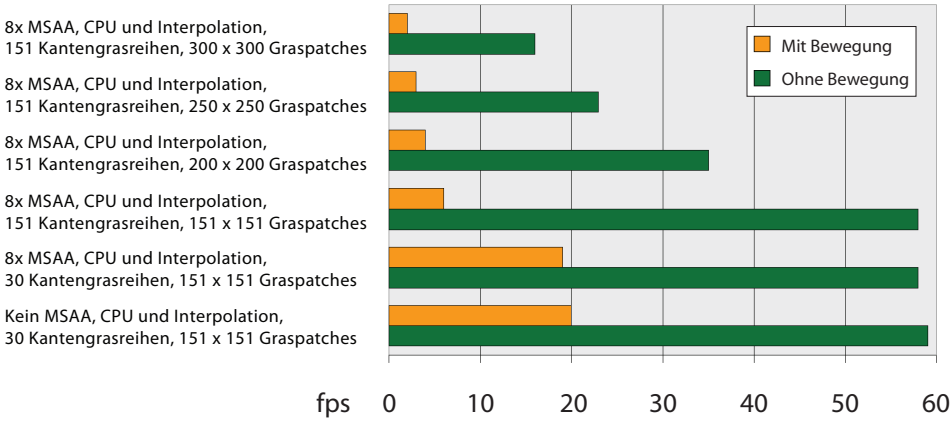

Abbildung 37: Gesamtperformance der Simulation in fps in Abhängigkeit verschiedener Einstellungen (Bildschirmauflösung: 800x600)

dadurch detailliert wiedergegeben werden. Mit Hilfe eines Texturatlasses können zudem mehrere Varianten von Gras ohne großen zusätzlichen Aufwand oder Performanceverlust genutzt werden, da wie bei einer einzelnen Grastextur nur eine `GL_TEXTURE_2D` gebunden werden muss – nur die Texturbreite und -höhe ist größer. Der Sichtwinkel ist allerdings entscheidend für die Darstellungsqualität, da eine Ansicht von oben schnell die Grasstruktur sichtbar werden lässt. Hierfür müsste ein zusätzlicher Billboardeffekt für die Texturquads hinzugefügt werden, damit sich das Gras zum Betrachter neigt, oder eine verbesserte Bodentextur zum Einsatz kommen.

Level-of-Detail-System

Das Level-of-Detail-System ermöglicht die Illusion einer unendlichen Wiese. Die Detailstufenübergänge sind bei zu kleiner Anzahl der Grasreihen deutlich zu erkennen, insbesondere der Übergang zwischen Kantengras und Bodentextur ist verbesserungswürdig. Weitere Probleme ergeben sich bei starken Höhenunterschieden, an denen auf Grund des Blickwinkels neben den Übergängen auch die Graspatches zu sehen sind (siehe Abbildung 38). Diese werden wie bei ebenem Terrain parallel zur xz-Ebene aufgestellt und schweben daher möglicherweise zum Teil in der Luft. Eine Anpassung an den Hügel würde allerdings bedeuten, dass die Quads bei Blick von der Ebene auf den Hügel visuell verschwinden. Zusätzliche Grasquads in mehrere Richtungen wären möglicherweise eine Lösung. Als besonders aufwändig stellte sich die Aktualisierung des Level-of-Detail-Systems bei Bewegung des Betrachters heraus. Das Potenzial, die Performance durch bessere Datenstrukturen zu erhöhen, ist groß (siehe Kapitel 15). Die Nutzung eines Level-of-Detail-Systems hat sich prinzipiell aber vor allem wegen des Partikelsystems gelohnt. Ohne dieses wäre die Wiese auf einen vordefinierten

Abbildung 38: Große Höhenunterschiede offenbaren das Level-of-Detail-System und die Texturquads: Blick auf den Berg (links) und Blick runter vom Berg (rechts), jeweils aus First-Person-Sicht

Bereich beschränkt gewesen. Zwar können Partikelsysteme schon aus sehr vielen Partikeln bestehen (siehe [Lat04]), dies bezieht sich allerdings auf reine Partikelsysteme ohne dynamische Starrkörper.

Kollisionsbehandlung

Kollisionen von Partikeln mit beweglichen Starrkörpern stellen heute noch ein Problem dar. In [Har09] werden daher Starrkörper aus Partikeln zusammengebaut um eine reine Partikelsimulation zu erhalten. Dadurch ist die Beschleunigung durch die GPU möglich, die Millionen von Partikeln parallel berechnen kann. Für Starrkörper gibt es auf der GPU noch keine allgemeine Lösung [Har09] – auch Nvidia PhysX berechnet die Rigid Bodies auf der CPU. Der Transfer der Partikeldaten von der CPU zur GPU und zurück erwies sich als zu performancekritisch. Daher stellt die Nutzung der GPU in dieser Diplomarbeit keine Beschleunigung, sondern eine Verlangsamung, dar. Dennoch ließen sich auch auf der CPU viele Partikel (siehe Kapitel 13) berechnen. Durch die weitere Reduzierung der Anzahl der Partikel durch die Nutzung einer Interpolationsstrategie konnte die Performance gesteigert werden, ohne dass die Animationsqualität darunter leidet. Weiterhin ermöglicht die Verwendung eines Partikelsystems grundsätzlich jegliche Arten von Kollisionskörpern.

Die Grasbewegung

Die Verschiebung der Vertices der Texturquads zur Grasbewegung übertraf die Erwartungen. Trotz der bloßen Verzerrung der Quads konnte ein glaubwürdiges Ergebnis geliefert werden. Eine zu große Kürzung der Textur bei Kollisionen erzeugt allerdings Probleme bei der Darstellung, da die Quads deutlich zum Vorschein kommen. Weitere Unterteilungen des

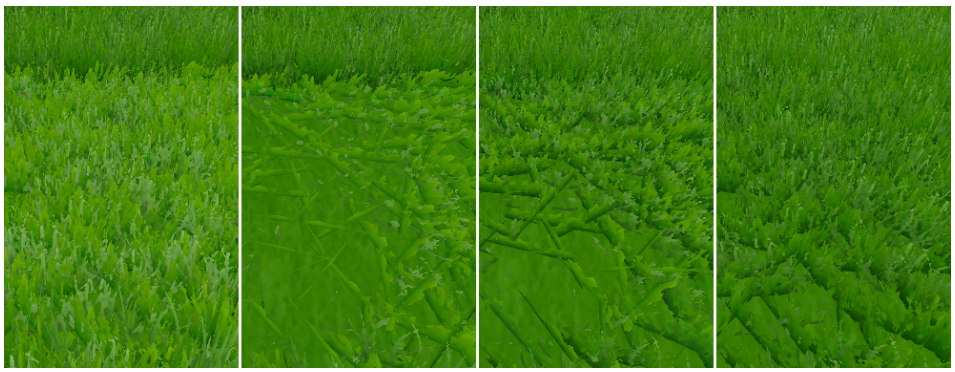

Abbildung 39: Gleichmäßige Neigung des Grases aus einer Richtung: direkt nach der Kollision mit einer Box (links, Ruhephase) bis kurz vor der vollständigen Wiederherstellung der Ausgangsposition (rechts). Das Gras bewegt sich im Bild von unten nach oben.

Texturquads waren nicht nötig. Durch die Nutzung des Texturatlasses kann sehr einfach auf eine spezielle Textur zur Hervorhebung der Kollision (in diesem Fall hellere Versionen der Gräser) umgeschaltet werden. Für eine korrekte Kollisionsbehandlung muss der Abstand der Partikel zum Terrain, also die Constraintdistanz, größer als die Höhe der in der Szene enthaltenen Objekte sein, damit das Objekt nicht unter die Partikel gerät und somit für die Kollisionserkennung nicht erfassbar wäre. Eine weitere Eigenheit des verwendeten Partikelsystems bzw. der verwendeten Kollisionsbehandlung ist die Durchdringung der Partikel in das Terrain in Richtung $(0, 1, 0)$. Dadurch wird zwar eine längere Ruhephase simuliert, gleichzeitig kann dieses Verhalten aber vor allem bei dem Würfelkollisionsprimitiv dazu führen, dass alle Partikel in gleicher Richtung wieder nach oben streben. Dadurch bewegen sich zeitgleich mehrere Texturquads aus der gleichen Richtung kommend in die senkrechte Position zurück, was dazu führt, dass die Texturquads aus einer bestimmten Richtung deutlich zum Vorschein kommen (siehe Abbildung 39).

15 Verbesserungsmöglichkeiten

Bereits im vorherigen Kapitel wurden einige Nachteile und mögliche Verbesserungsmöglichkeiten angesprochen. In diesem Kapitel wird auf konkrete Methoden zur Verbesserung der größten Schwachstellen der Implementation eingegangen.

Texturen

Höher auflösende Grastexturen könnten visuelle Ungenauigkeiten beheben. Besonders bei näherer Betrachtung fallen die Treppeneffekte innerhalb

der Textur auf Grund des Alphatests auf. Alpha-To-Coverage konnte nur geringfügig für Abhilfe schaffen. Die Veröffentlichung von [Gre07] zeigt allerdings, dass durch die Nutzung eines speziellen Alphakanals innerhalb der Textur die Ränder der Objekte innerhalb der Textur geglättet werden können. Dadurch wäre die Darstellungsqualität von semi-transparenten Texturen mit Alphablending kaum mehr zu unterscheiden.

Verschwindende Texturquads

Das Problem des Verschwindens der Texturquads aus einem bestimmten Blickwinkel (senkrecht von oben auf die Kante des Quads) wäre nur durch das Hinzufügen eines Billboardeffekts lösbar. In der *Unigine Demo* wird ein Verfahren angewendet, das die Texturquads dezent bewegt, d. h. nicht aus allen Blickrichtungen und nicht komplett um ihre eigene Achse. Da zusätzlich die Partikel die Vertices der Grasquads verschieben, müsste eine solche Vorgehensweise auf ihre Anwendbarkeit getestet werden.

Beleuchtung

Der Einsatz von Screen Space Ambient Occlusion hat gezeigt, dass Post-Processing-Effekte auch mit semi-transparenten Texturen ohne Probleme funktionieren. Neben der dadurch gewonnenen Verschattung könnte die Beleuchtung entsprechend verbessert werden: Da die Daten (Normalen, Positionen, etc.) bereits in einem Frame Buffer Object gespeichert sind, wäre die Nutzung von Deferred Lighting einfach zu bewältigen. Die Beleuchtung, die in der bisherigen Implementation per OpenGL-Standard-Beleuchtung bzw. mittels eines einfachen Shaders programmiert wurde, könnte so wesentlich verbessert werden.

Level-of-Detail-System

Das Level-of-Detail-System ist als solches stark ausbaufähig. Es war als *die* Möglichkeit gedacht, überhaupt eine große Wiese zu simulieren. Um eine derartige Wiese ohne Level-of-Detail-System zu verwirklichen, wäre eine sehr viel größere Menge an Texturquads notwendig gewesen – daher kann das Level-of-Detail-System auch gleichzeitig als eine Beschleunigungstaktik angesehen werden. Leider stellte sich heraus, dass die Aktualisierung des Systems in Verbindung mit dem Partikelsystem und den Vertex Buffer Objects ein „Performanceloch“ ist. Da die Daten für den VBO in einer festgelegten Struktur vorliegen müssen und komplexere Datenstrukturen, wie sie für das Level-of-Detail-System notwendig waren, ungeeignet sind, müssen große Menge der Daten bei Bewegung der Kamera kopiert werden und ganze STL-Vektoren neu geschrieben werden. In einer performancekritischen Echtzeitanwendung ist dies nicht akzeptabel. Zwar kommt man

prinzipiell um das Uploaden neuer Positionsdaten an den VBO nicht herum, evtl. könnte man aber die Menge der neuen Daten durch die Nutzung von Indizes für die Buffer verringern. Für das Level-of-Detail-System bliebe anstelle von STL-Vektoren und STL-Listen nur die Nutzung gänzlich anderer Datenstrukturen wie z. B. Bäumen übrig. Diese würden einen Komplettumbau des Level-of-Detail-Systems erfordern und konnten daher in dieser Diplomarbeit nicht berücksichtigt werden. Ein weiterer Kritikpunkt am Level-of-Detail-System ist der Übergang zwischen der letzten und mittleren Detailstufe. Die Bodentextur wurde per Hand erstellt, sollte in Zukunft aber zur Laufzeit aus der Simulation heraus erstellt werden. In diesem Zusammenhang muss aber eine Lösung für das Problem der verschwindenden Texturquads gefunden werden um eine vernünftige Textur zu erhalten.

Physik- bzw. Partikelsystem

Der Vorteil, ein Partikelsystem schnell auf der GPU berechnen zu können, konnte nicht ausgespielt werden. `glReadBuffer` und `glReadPixels` erwiesen sich als viel zu langsam, trotz Verkleinerung der Texturgrößen durch Interpolation. Die einzige Möglichkeit, um den Geschwindigkeitsverlust durch das Hoch- und Runterladen der Daten zu vermeiden, wäre die Transferierung aller mit den Partikeln zusammenhängenden Berechnungen auf die GPU. Dazu zählt die Integration der Starrkörper, die komplette Kollisionserkennung und -behandlung und die Bewegung der Grasquads. Wie in Kapitel 14 bereits angesprochen, könnten die Starrkörper für die GPU als Partikel dargestellt werden um wiederum eine reine Partikelsimulation zu erhalten. Zusätzlich müsste die Positionsänderung der Vertices der Grasquads mit einem Shader durchgeführt werden. Dadurch würde auch die Aktualisierung der VBOs entfallen. Der von üblichen Partikelsystemen gewählte Ansatz, Billboards an den Positionen der Partikel zu rendern, ist auf die Grassimulation nicht anwendbar, da sich nicht das komplette Grasbüschel bewegen darf. Die implementierte GPU-Berechnung verwendete OpenGL und GLSL. Für künftige GPGPU Anwendungen werden die speziell dafür entwickelten Programmierschnittstellen wie Nvidias CUDA oder OpenCL wichtig und könnten an solch einer Aufgabenstellung getestet werden.

In dieser Diplomarbeit wurde die Cyclon-Physik-Engine verwendet um Zugriff auf die Algorithmen zu haben um sie gegebenenfalls ändern zu können. Sie wurde so konzipiert, dass sie im Rahmen des Buches von Millington [Mil07] gut erklärbar und einfach zu verstehen ist. Daher ist sie nicht notwendigerweise schnell oder (speicher-)effizient. Der Einsatz einer *professionelleren* Physik-Engine wie Nvidia PhysX wäre im Hinblick auf dessen potentiellen Geschwindigkeitszuwachs und stetige Weiterentwicklung zu überlegen. Zudem ermöglicht sie alle Arten von Kollisionsobjekten und in Zukunft möglicherweise die GPU-beschleunigte Starrkörpersimulation.

Weitere Beschleunigungsverfahren

Zusätzlich zu den implementierten Beschleunigungsverfahren (VBOs, Texturatlas, Level-of-Detail-System, Interpolation, GPU-Nutzung) gibt es noch einige weitere Optionen zur Performancesteigerung. Cullingverfahren (View-Frustum-Culling, Occlusion-Culling) wären naheliegend um viele Grasbüschel vom Rendering auszuschließen. Auch Geometry Instancing (siehe [Car05]) wurde bereits verwendet um viele gleichartige Gräser mit Hilfe der Grafikkartenhardware beschleunigt darzustellen. Dieses Verfahren kann zusätzlich durch Culling mittels Transform Feedback erweitert werden [Rák10].

16 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde eine Echtzeitanwendung entwickelt, die Gras darstellt und zugleich äußere Einflüsse wie Wind oder Kollisionen auf das Gras berücksichtigt. Ein Level-of-Detail-System ermöglicht die Darstellung und Physiksimulation einer großen Wiese in drei Stufen. Semi-transparente Texturquads zur Darstellung und Partikel mit Constraints an der Stelle der Graspatches stellen die feinste Detailstufe in allernächster Nähe zum Betrachter dar. Die Partikel, die nur auf der feinsten Detailstufe verwendet werden, werden mittels einer Physik-Engine berechnet und reagieren auf verschiedene Starrkörper. Die zweite Detailstufe besteht ebenfalls aus Texturquads, die um das erste Detaillevel herum generiert werden. Diese sind jedoch spärlicher verteilt, sehen auf Grund der Distanz zum Betrachter allerdings identisch mit denen aus der ersten Stufe aus. Als letzte Repräsentationsform von Gras kommt eine einfache Bodentextur zur Anwendung. Durch dieses Level-of-Detail-System ist es möglich eine unendlich große Wiese zu simulieren, da die Gräser zur Laufzeit generiert werden. Ein umfangreiches Benutzerinterface ermöglicht die Veränderung einiger Simulationsparameter in Echtzeit und die Generierung verschieden großer Rasenflächen. Zusätzlich lassen sich verschiedene Objekte in der Simulationsumgebung bewegen um die Kollisionsbehandlung testen zu können.

Die Evaluation der Darstellungs- und Animationsqualität hat ergeben, dass die Grassimulation teilweise ähnliche Resultate wie andere Veröffentlichungen hervorbringt. Einige aus der Realität entlehnte Beobachtungen, wie z. B. die geringe Bewegung von (kurzem) Gras auf Grund von Wind bzw. die Lokalität der Kollisionsbehandlung, wurden für die Simulation übernommen und konnten entsprechend mit der Natur und anderen Implementationen verglichen werden. Trotz der einfachen physikalischen Berechnung der Gräser mittels Partikel konnte eine glaubwürdige Simulation erzeugt werden.

Da die Anforderungen an eine Echtzeitanwendung hinsichtlich ihrer Performance hoch sind, wurden verschiedene Beschleunigungsmöglichkeiten implementiert. Für die Darstellung wurden Vertex Buffer Objects verwendet um die enorme Menge an Grasquads darstellen zu können. Als Schwachstelle der Vertex Buffer Objects hat sich der Aktualisierungsvorgang der Daten erwiesen. Dieser könnte durch eine verbesserte Datenstruktur des Grases wesentlich entschärft werden. Zur Beschleunigung des Partikelsystems wurde die Berechnung auf die GPU ausgelagert und das Ergebnis interpoliert. Die Berechnung auf der GPU erwies sich auf Grund des langsamen Up- und Downloads der Daten zwischen CPU und GPU eher als Bremse denn als Beschleunigung. Um die Physik auf der GPU zu beschleunigen, müsste die komplette Grassimulation auf der GPU berechnet werden, damit die Aktua-

lisierung der Daten für die CPU entfallen kann. Da die GPU ideal geeignet für viele parallele unabhängige Berechnungen ist, könnte sich eine solche Erweiterung für die Performance als nützlich erweisen, würde allerdings auf Grund der Starrkörper äußerst aufwändig werden. Die Interpolation hingegen erzielte eine große Performancesteigerung ohne sichtbaren negativen Einfluss auf die Bildqualität.

Um die Grasdarstellung weiter zu verbessern, könnten höher aufgelöste, detaillierte Texturen in mehreren Variationen verwendet werden. Um die Kanten des Grases zu glätten, könnten die Schritte aus [Gre07] mit eingebracht werden, was eine aufwändigere Sortierung der Grasbüschel für weiches Alphablending überflüssig macht. Möglicherweise ließe sich die Kollisionsreaktion der Grasquads durch deren Unterteilung für nahe Betrachterperspektiven weiter verbessern. Es existieren zudem weitere Beschleunigungsmöglichkeiten wie z. B. View-Frustum-Culling um weniger Grasbüschel zu visualisieren und die Performance zu erhöhen. Da bereits die eingesetzten Verfahren eine Echtzeitsimulation zulassen und das Konzept an sich somit realisierbar ist, sind für weitere Optimierungen kaum Grenzen gesetzt.

Auch in Zukunft werden Grassimulationen eine bedeutende Rolle in Echtzeitanwendungen spielen. Die Simulation von Physik in Computerspielen nimmt zunehmend eine wichtige Rolle ein um die Immersion in virtuelle Welten zu erhöhen. Wohingegen in den Anfängen oftmals die Levels in engen geschlossenen Räumen stattfanden, sind weitläufige Naturszenen heute der Inbegriff von Freiheit und auf Grund der heutigen schnellen Grafikkarten visualisierbar. Interaktive Wiesen sind daher der logische nächste Schritt, den es in zukünftigen Echtzeitanwendungen zu implementieren gilt. Aktuelle Veröffentlichungen wie von [CJ10] oder [ORSK09] zeigen zusammen mit dieser Diplomarbeit, dass es bereits Möglichkeiten zur Simulation von Gras mit Behandlung von Kollisionen gibt, die es nun gilt in größeren interaktiven virtuellen Welten erlebbar zu machen.

Abbildungsverzeichnis

1 Gras im Animationsfilm *Big Buck Bunny* 2
2 Geometriebasiertes Verfahren (für die feinste Detailstufe, nach [Bou08]) 4
3 Level-of-Detail-System mit Billboards und Bodengrastextur aus *Unigine Engine Benchmark* 2010 5
4 Geschachtelte Texturquads aus *Codecreatures Demo* [Pel07] . 6
5 Virtuelle Bilboards: Struktur ähnlich zu Relief Mapping (links) und Resultate (rechts) mit Hervorhebung der Struktur (oben) [HWJ07] 7
6 Level-of-Detail-System, welches in [Bou08] zum Einsatz kommt 10
7 Fotografiertes Gras mit erkennbaren Glanzpunkten und Verschattungen 11
8 Volumenbasiertes Renderingverfahren mittels Shells – hier das Verfahren von [Ban06] mit zwei Topologien am Beispiel einer *Texcell* in der Mitte (oben: *Spring-stick topology*, unten: *Prism topology*) 15
9 Nvidia PhysX (Beispiel aus dem SDK, *Lesson405*) 20
10 Kontakt-Koordinatensysteme und Vektoren (nach [Mil07]) . 36
11 Level-of-Detail-System: Oben aus First-Person-Sicht, unten aus der Vogelperspektive (rechts jeweils zur Verdeutlichung mit Anzeige der Partikel) 44
12 Benötigte Vektoren bei Kamerabewegung, wobei $\overrightarrow{side}$ ein Einheitsvektor in die entsprechendene Richtung (top, right, bottom, left) ist. 45
13 Vergleich Graspositionierung ohne (links) und mit (rechts) Zufälligkeitsfaktor 47
14 Update der STL-Vektoren: Löschen (hellgrün) und Hinzufügen (dunkelgrün) von Grasreihen für die vier Bewegungsfälle 49
15 Struktur des Kantengrases 50
16 Aktualisierung des Kantengrases in Pfeilrichtung in vier Schritten. Im Zentrum befindet sich das Gras der feinsten Detailstufe. Die grünen ungefüllten Quadrate stellen die neu zu generierenden Patches dar, die grün gefüllten sind die Eckpositionen. Die schwarz schraffierten Patches müssen gelöscht werden. 53
17 Screen Space Ambient Occlusion: Ohne SSAO (links), Ambient Occlusion Wert (Mitte), mit SSAO (rechts), jeweils mit diffuser Shader-Beleuchtung 55
18 Alpha To Coverage: an (links) und aus (rechts). Dithering wird am Grashorizont sichtbar. 56

19 Grobe Kollisionserkennung von Starrkörpern (schwarz). Rote Punkte sind die Eckweltkoordinaten, die roten Quadrate die dazugehörigen Graspatches (bzw. deren Indizes). Alle orangenen Patches sind potenzielle Kollisionskandidaten. . 62
20 Kollisionsbehandlung von einem Starrkörper, der über die Wiese bewegt wurde 63
21 Aktualisierung des Grases bei Neigung entsprechend des Partikels 64
22 Texturatlas mit Indexnummern (von normal nach hell: rot, grün, blau, orange) 65
23 Interpolationsfälle für die roten Graspatches: Keine Interpolation, Interpolation zwischen vier oder zwei Partikeln und Randfälle 67
24 Analogie zwischen Partikeln und Texturen und Aufbau der Texturen zur GPU-Berechnung 68
25 Screenshot der Grassimulation mit GUI 71
26 Rasenmäher und teilweise gemähtes Gras (mit aktiviertem SSAO) 73
27 Gras erholt sich von einer Kollision mit einer Kiste 73
28 Reduzierung der Gräser bei großer Entfernung vom Betrachter und Alphablending in *Rendering Countless Blades of Waving Grass* [Pel07] 76
29 Vogelperspektive (1. und 3. Bild) und First-Person-Perspektive (2. und 4. Bild) im Vergleich: *SimGrass* (links) und [Ban06] (rechts) 77
30 Vergleich des Level-of-Detail-Systems mit [Bou08] 78
31 Kollisionsbehandlungen im Vergleich (von links oben im Uhrzeigersinn: *SimGrass*, [ORSK09], [CJ10], [GPR+03]) 79
32 Vergleich mit der Realität: mit SSAO (links), ohne SSAO (mittig), Referenz (rechts) 80
33 Kollision in der Realität: mit Objekten (links) und kurz nach Entfernung der Objekte (rechts, rot markiert die Lage der Objekte) 81
34 Links: Zeitmessung für die GPU und die CPU Integration. Rechts: Ausführungsgeschwindigkeit des Fragment-Shaders für die Integration und Geschwindigkeit der `Readpixels`- und `Readbuffer`-Operationen (auf zwei Texturen) 83
35 Links: CPU-Integration mit und ohne Interpolation. Rechts: Update der Grasneigungen mit und ohne Interpolation. 83
36 Zeitmessung für die gesamte Applikation (ohne Physiksystem) bei Bewegung des Betrachters in Richtung einer Ecke des Grasfelds 85
37 Gesamtperformance der Simulation in fps in Abhängigkeit verschiedener Einstellungen (Bildschirmauflösung: 800x600) 86

38 Große Höhenunterschiede offenbaren das Level-of-Detail-System und die Texturquads: Blick auf den Berg (links) und Blick runter vom Berg (rechts), jeweils aus First-Person-Sicht 87
39 Gleichmäßige Neigung des Grases aus einer Richtung: direkt nach der Kollision mit einer Box (links, Ruhephase) bis kurz vor der vollständigen Wiederherstellung der Ausgangsposition (rechts). Das Gras bewegt sich im Bild von unten nach oben. 88

Literatur

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[And10] Johan Andersson. 5 major challenges in interactive rendering. `http://publications.dice.se/attachments/Siggraph10-BPS-5_Major_Challenges.ppt`, 2010. [Online, abgerufen am 14.09.2010].

[Bak03] Brook Bakay. Animating and lighting grass in real-time. Master's thesis, University of British Columbia, April 2003.

[Ban06] Sven Banisch. Making grass and fur move. *Journal of WSCG*, 14, 2006.

[BCF+05] S. Behrendt, C. Colditz, O. Franzke, J. Kopf, and O. Deussen. Realistic real-time rendering of landscapes using billboard clouds. In M. Alexa and J. Marks, editors, *Computer Graphics Forum*, volume 24, pages 507–516. Eurographics, 2005.

[BLH02] Brook Bakay, Paul Lalonde, and Wolfgang Heidrich. Real time animated grass. In *Proceedings of Eurographics (short paper)*. Eurographics, 2002.

[Bou08] Kévin Boulanger. *Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting*. PhD thesis, University of Rennes, France, 2008.

[Car05] Francesco Carucci. Inside geometry instancing. *GPU Gems 2*, 2005.

[CCDH05] Carsten Colditz, Liviu Coconu, Oliver Deussen, and Hans-Christian Hege. Real-time rendering of complex photorealistic landscapes using hybrid level-of-detail approaches. In *First publ. as: Paper / Conference for Information Technologies in Landscape Architecture, 2005*. Universität Konstanz, 2005.

[CJ10] Kan Chen and Henry Johan. Real-time continuum grass. In *VR*, IEEE Virtual Reality 2010, 2010.

[dB00] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping. `http://www.flipcode.com/archives/Fast_Terrain_Rendering_Using_Geometrical_MipMapping.shtml`, Oktober 2000. [Online, abgerufen am 14.09.2010].

[DN04] Philippe Decaudin and Fabrice Neymet. Rendering forest scenes in real-time. In H. W. Jensen and A. Keller, editors, *Rendering Techniques '04 (Eurographics Symposium on Rendering)*, pages 93–102. Eurographics, Juni 2004.

[DN09] Philippe Decaudin and Fabrice Neyret. Volumetric billboards. *Comput. Graphics Forum*, 28:2079–2089, Dezember 2009.

[GCF01] Thomas Di Giacomo, Stéphane Capo, and François Faure. An interactive forest. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 65–74, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[GPR+03] Sylvain Guerraz, Frank Perbet, David Raulo, François Faure, and Marie-Paule Cani. A procedural approach to animate interactive natural sceneries. In *CASA '03: Proceedings of the 16th International Conference on Computer Animation and Social Agents (CASA 2003)*, page 73, Washington, DC, USA, 2003. IEEE Computer Society.

[Gra10] Silicon Graphics. Opengl 2.1 reference pages – glsamplecoverage. `http://www.opengl.org/sdk/docs/man/xhtml/glSampleCoverage.xml`, 2010. [Online, abgerufen am 14.09.2010].

[Gre07] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 9–18, New York, NY, USA, 2007. ACM.

[Har09] Takahiro Harada. Parallelizing the physics pipeline: Physics simulations on the gpu, physics for programmers. *Game Developers Conference*, 2009.

[HWJ07] Ralf Habel, Michael Wimmer, and Stefan Jeschke. Instant animated grass. *Journal of WSCG*, 15(1–3):123–128, Januar 2007. ISBN 978-80-86943-00-8.

[KK89] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *SIGGRAPH Comput. Graph.*, 23(3):271–280, 1989.

[Lat04] Lutz Latta. Building a million particle system. pages 54–60. Game Developers Conference, 2004.

[Mil07] Ian Millington. *Game Physics Engine Development (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[Mit07] Martin Mittring. Finding next gen: Cryengine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 97–121, New York, NY, USA, 2007. ACM.

[Mül10] Stefan Müller. Orientierung. `http://userpages.uni-koblenz.de/~cg/ss10/ansim/04_orientierung.pdf`, 2010. [Online, abgerufen am 13.09.2010].

[Ney98] Fabrice Neyret. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55–70, 1998.

[ORSK09] J. Orthmann, C. Rezk-Salama, and A. Kolb. GPU-based Responsive Grass. *Journal of WSCG*, 17:65–72, 2009.

[OW10] OpenGL-Wiki. Vertex specification best practices. `http://www.opengl.org/wiki/VBO_-_more`, 2010. [Online, abgerufen am 13.09.2010].

[PC01] Frank Perbet and Maric-Paule Cani. Animating prairies in real-time. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 103–110, New York, NY, USA, 2001. ACM.

[Pel07] Kurt Pelzer. Rendering countless blades of waving grass. *GPU Gems*, pages 107–121, 2007.

[Per05] Emil Persson. Alpha to coverage. `http://www.humus.name/index.php?page=3D&ID=61`, Juni 2005. [Online, abgerufen am 13.09.2010].

[Rák10] Daniel Rákos. Instance culling using geometry shaders. `http://rastergrid.com/blog/2010/02/instance-culling-using-geometry-shaders/`, Februar 2010. [Online, abgerufen am 17.09.2010].

[SKP05] Musawir A. Shah, Jaakko Kontinnen, and Sumanta Pattanaik. Real-time rendering of realistic-looking grass. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 77–82, New York, NY, USA, 2005. ACM.

[Tha02] Ulrich Thatcher. Rendering massive terrains using chunked level of detail control. `http://tulrich.com/geekstuff/chunklod.html`, April 2002. [Online, abgerufen am 14.09.2010].

[Whi08] Matthew White. Real-time optimally adapting meshes: terrain visualization in games. *Int. J. Comput. Games Technol.*, 2008:1–7, 2008.

[Wik10a] Wikipedia. Orthonormalbasis — wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Orthonormalbasis&oldid=78215051, 2010. [Online, abgerufen am 26.08.2010].

[Wik10b] Wikipedia. Parallaxe — wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Parallaxe&oldid=77550785, 2010. [Online, abgerufen am 28.08.2010].

[Wik10c] Wikipedia. Procedural generation — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Procedural_generation&oldid=380934015, 2010. [Online, abgerufen am 26.08.2010].

[Wik10d] Wikipedia. Stoß (physik) — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=Sto%C3%9F_(Physik)&oldid=77848398, 2010. [Online, abgerufen am 21.09.2010].