FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Interaktives Gras Rendering in Echtzeit unter Verwendung moderner OpenGL Methoden

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

BSc Klemens Jahrmann
Matrikelnummer 0826080

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 1. Jänner 2001

Klemens Jahrmann

Michael Wimmer

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Interactive Grass Rendering in Real Time Using Modern OpenGL Features

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

BSc Klemens Jahrmann
Registration Number 0826080

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Vienna, 1^{st} January, 2001

Klemens Jahrmann   Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

BSc Klemens Jahrmann
Bukovicsgasse 33, 1220 Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Jänner 2001

____

Klemens Jahrmann

Danksagung

Zunächst möchte ich meinem Betreuer Michael Wimmer danken, der mich über den gesamten Entstehungsprozess dieser Diplomarbeit begleitet hat. Er gab mir die Möglichkeit, mein Interesse für das Gras Rendering noch weiter zu vertiefen und unterstützte mich in weiterer Folge zu allen erdenklichen Tages- und Nachtzeiten mit Verbesserungsvorschlägen. Die Zusammenarbeit empfand ich als sehr angenehm und die schnellen Rückmeldungen haben die Finalisierung dieser Arbeit erst möglich gemacht.

Weiters möchte ich meinen Kolleginnen und Kollegen vom Institut für Computergrafik danken, die mir bei Problemen stets mit Rat und Tat zur Seite gestanden sind. Ein besonderer Dank hierbei gilt meinem langjährigen Freund und Kollegen Bernhard Steiner, der als Ansprechperson immer für mich da war.

Ebenso dankbar bin ich für die Unterstützung meiner Eltern, die mir die Voraussetzungen für ein sorgenfreies Studium geschaffen haben, sowie meinem Bruder, der mir bereits früh das Fachgebiet der Informatik nahe gebracht hat. Nicht zuletzt möchte ich mich für das wiederholte Korrekturlesen der verschiedenen Version meiner Arbeit bei meiner Freundin Elisabeth Sperr bedanken. Besonders als motivierende Kraft und seelische Unterstützung stand sie stets hinter mir.

Acknowledgements

First of all, I would like to thank my supervisor Michael Wimmer, who supported me during the process of writing this thesis. He gave me the opportunity to go deeper into my favorite topic of grass rendering. Furthermore, he supported me with suggestions for improvement at day- and nighttime. I really enjoyed the collaboration. I also want to add that only his extremely fast answers and corrections made the completion of this thesis possible.

In second place, I would like to thank my colleagues at the institute of computer graphics for their help and discussions in complicated situations. In this regard, my special thanks are directed to my longtime friend and colleague Bernhard Steiner. He was there for me anytime I needed help.

In addition, I am very thankful for the support from my parents, who laid the foundations for my time at university. Another important inspiration for my studies was my brother, whom I want to thank for introducing me to the field of computer science. Last but not least, I would like to thank my girlfriend Elisabeth Sperr for her enormous effort on proofreading the different stages of this thesis. She supported me especially in matters of motivation and encouragement.

Kurzfassung

Süßgräser sind ein wichtiger Bestandteil der weltweiten Vegetation und kommen in allen Klimazonen vor. Aus diesem Grund befindet sich in so gut wie jedem Außenszenarium Gras. Bis heute gibt es nur wenige Algorithmen zum Rendern von Gras in Echtzeit aufgrund der hohen geometrischen Komplexität. Daher arbeiten die meisten Algorithmen mit Billboards oder anderen bildbasierte Verfahren, welche jedoch Probleme haben, Animationen und physicalische Interaktionen darzustellen. Ein weiterer Nachteil von bildbasierten Verfahren ist jener, dass diese oft Artefakte beim Rendern aus bestimmten Blickwinkeln aufweisen, da diese zweidimensionale Bilder in den dreidimensionalen Raum einbetten.

In dieser Arbeit stellen wir ein Verfahren vor, mithilfe dessen man Gras in kompletter geometrischer Repräsentation in Echtzeit rendern kann. Der Algorithmus ist sehr allgemein geschrieben und kann leicht verändert beziehungsweise erweitert werden, damit er sich gut an die meisten Szenarien anpasst, in denen vom Rendern von Gras oder grasähnlichen Vegetationen handelt.

Abstract

Grass species are an important part of vegetation all over the world and can be found in all climatic zones. Therefore, grass can be found in almost all outside scenarios. Until today, there are only few sophisticated algorithms for rendering grass in real time due to the high amount of geometrical complexity. As a result, most algorithms visualize grass as a collection of billboards or use other image-based methods, which have problems dealing with animation or physical interaction. Another disadvantage of image-based methods is that they often have artifacts when viewed from specific angles, because they are just two-dimensional images embedded in three-dimensional space.

In this thesis we will introduce a fully geometric approach of grass rendering working at interactive framerates. The algorithm is very generic and is able to be adjusted and extended in various ways in order to be applicable to most scenarios of rendering grass or grass-like vegetation.

Contents

Kurzfassung **xi**

Abstract **xiii**

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem statement . . . 2
- 1.3 Contributions . . . 3
- 1.4 Structure of work . . . 4

2 State of the Art **5**
- 2.1 Image-based approaches . . . 5
- 2.2 Geometrical approaches . . . 7
- 2.3 Hybrid approaches . . . 9

3 Overview **11**
- 3.1 Conceptional structure . . . 11
- 3.2 Basic algorithm . . . 12

4 Preprocessing **15**
- 4.1 Definition of a blade of grass . . . 15
- 4.2 Distribution . . . 17
- 4.3 Patch generation . . . 24

5 Physical Behavior **31**
- 5.1 Force Map . . . 32
- 5.2 Stiffness . . . 33
- 5.3 Gravity . . . 34
- 5.4 Wind . . . 35
- 5.5 Ground collision . . . 39
- 5.6 Calculation of v_1 . . . 40
- 5.7 Conservation of length . . . 40
- 5.8 Sphere collision . . . 42
- 5.9 Complex object collision . . . 44

6 Culling Methods **47**
6.1 High-level view-frustum culling 47
6.2 Low-level culling 48

7 Rendering **57**
7.1 Vertex-shader stage 57
7.2 Tessellation shader stage 59
7.3 Blade geometry 60
7.4 Fragment-shader stage 64

8 Implementation Details **67**
8.1 Application structure 67
8.2 Preprocessing details 68
8.3 OpenGL features 69
8.4 Maximum inner sphere 72

9 Results **77**
9.1 Patch generation results 77
9.2 Scenes and evaluation 82
9.3 Discussion 89

10 Conclusion and Future Work **95**

List of Figures **99**

List of Tables **104**

List of Algorithms **105**

Bibliography **107**

CHAPTER 1

Introduction

1.1 Motivation

In modern times, interactive media have become more and more important in everyday life. The major difficulty of interactive applications is that each presented image has to be calculated and drawn in real time according to the user's input. Therefore, real-time rendering often needs to fall back on fast approximations, especially for rendering complex scenes. However, the fast development of graphics hardware and graphics programming interfaces enables real-time applications to achieve a higher grade of photo-realism. In addition, this development offers a high range of realistic scenes to be rendered in real time.

Outdoor scenes are often considered more complex than indoor scenes, because of the high complexity of the vegetation. Grass represents an important part of the vegetation. According to the report on grasslands of the Food and Agriculture Organization of the United Nations [FotUN], up to forty percent of the earth's land area is covered by grasslands, including pastures and fodder crops. Therefore, grass is essential for a majority of outdoor terrains. Such terrains are traditionally represented as heightmaps in real-time applications. Heightmaps are flat surfaces that are vertically extruded according to a given image. However, the variety of terrains generated by a heightmap is limited. For example the generated terrain cannot have overlapping regions, holes or cavities. Therefore, instead of using heightmaps, modern applications often represent terrains as three-dimensional models.

The typical grass representation on such heightmaps or three-dimensional models are texture-based approaches. They include textured surfaces, aligned billboards or ray-traced texture slices. Thus, an improvement of the visual appearance of the grass representation can be mostly achieved by creating more realistic textures. The drawback of many texture-based techniques is the occurence of rendering artifacts when viewed from specific angles. This is due to the fact that they are two-dimensional representations embedded in the three-dimensional space.

Another drawback of traditional grass-rendering approaches is that they cannot react according to their environment. For example, they are not influenced by wind or other forces. This is problematic since interactivity is extremely necessary for the virtual experience of the user. This is made more concrete by Alison McMahan [McM03], who stated three important rules for a three-dimensional virtual reality to produce immersion to the viewer:

1. The user's expectations of the game or environment must match the environment's conventions.
2. The user's actions must have a non-trivial impact on the environment.
3. The conventions of the world must be consistent.

When analyzing these rules, none of them really depend on the photo-realism of the visual appearance. Therefore, a good grass-rendering technique needs to provide more features than well-designed textures and special effects. According to the first rule, the behavior of the blades of grass has to match their environment. If the weather of the virtual scene seems to be nice, the blades should bow in a gentle breeze with some turbulences and occasional gusts. Otherwise, if the virtual weather is dark and stormy, the blades should be whipped by the wind and even respond to raindrops. The second rule indicates that the user has to have an influence on the virtual environment. Therefore, the collisions between the blades of grass and the objects in the three-dimensional space have to be recorded and processed in an expected way. Finally, the third rule links the first two rules together for the whole virtual environment. To give an example, if a player is able to run his car over bushes leaving trails, he should also be capable of leaving trails in the grass. In addition, the strength of the wind bending the branches of trees should be the same for bending the blades of grass.

1.2 Problem statement

In recent times, more sophisticated grass-rendering techniques have been developed and traditional texture-based approaches have been enhanced. Nevertheless, no real-time algorithm is capable of both having a realistic appearance and being interactive with the environment at the same time until today. Some approaches tend towards having a realistic appearance [BPB09] but lack interactivity. Others draw grass represented by billboards, which have the ability to interact with the environment [JSK09]. However, the drawbacks of traditional billboard methods become even more visible when being distorted by collisions. Modern techniques focus on rendering each blade of grass as geometrical object. Some of them aim at showing realistic natural behavior of grass fields [JW13]. Due to the high geometrical complexity, they have to make use of coarse approximations to achieve interactive frame rates. Others specialize on the physical interaction of the blades of grass [FLHS15]. However, the grass field tends to looks artificial, because all blades of grass share a uniform appearance. In addition, most techniques that represent each blade of grass as geometrical object use the geometry of a single rectangular patch and render it multiple times over the scene. On the one hand, this reduces the memory footprint. On the other hand, the terrain is bound to be a flat surface or a heightmap, which does not correspond to the desires of modern applications.

1.3 Contributions

In this thesis we introduce a grass-rendering technique that solves the problems described above, and does so in real time. The realistic appearance is achieved by drawing each blade of grass as two-dimensional geometrical object embedded in three-dimensional space. In addition, interactivity is achieved by allowing the blades of grass to be influenced by natural forces and to collide with the environment. Furthermore, in contrast to most modern sophisticated techniques, which are designed to operate only on flat grass fields or heightmaps, our technique is able render grass fields of arbitrary shape and spatial alignment.

The proposed algorithm is based on our previous work [JW13], although only the main ideas of the rendering are still in common. Fast and accurate culling methods, performed on each blade of grass, enable the replacement of coarse approximations by physically based calculations. In order to match the rules of immersion by Alison McMahan [McM03], all influences are calculated for each blade of grass separately. Furthermore, the shape of a blade is defined by an analytic function, which allows having a more realistic and heterogenous appearance of a grass field. In comparison to previous and related work, the most important contributions of our method are stated below.

- In contrast to other grass rendering techniques, the blades of grass can be *distributed on arbitrary objects* and are not bound to flat surfaces or heightmaps. Thus, all algorithms have to be able to operate on blades of grass aligned in any direction in three-dimensional space. Different distribution algorithms allow having more realistic visual appearances of meadows in various scenarios.

- A *physical model* is introduced for each blade of grass to simulate natural effects. Those effects include the influence of *wind*, *gravity* and *collisions* with both simple and complex objects. In addition, physical properties like length and forces stay consistent for each blade of grass.

- The *shape* of a blade of grass is computed using analytic functions instead of providing an alpha texture. This enables accurate edges of the shape regardless of the distance from the blade of grass to the viewer.

- Thanks to *culling methods performed on each blade of grass*, only blades that are visible in the current frame and not hidden behind objects have to pass through the rendering pipeline. In addition, the culling methods are able to cull a certain fraction of the blades depending on the distance to the viewer. This is important, because rendering artifacts can appear if too many blades are projected onto the same pixels.

- The rendering of each blade of grass is performed on the basis of a *dynamic level-of-detail* approach using hardware tessellation.

1.4 Structure of work

The following Chapter 2 gives an overview to different grass rendering techniques, divided into image-based, hybrid and geometrical approaches.

An introduction to our rendering technique including the basic algorithm, the conceptional structure and the definition of a blades of grass is presented in Chapter 3.

In Chapter 4, we present the preprocessing steps, which are needed for the algorithm. This includes the distribution of the blades of grass over a three-dimensional object and the generation of grass patches from single blades of grass.

Chapter 5 explains the physical behavior of a single blade of grass and its implementation together with the necessary data structures. Among other, this includes the influence of wind and collisions with both simple and complex objects.

In order to lighten the workload of the graphics hardware, only the blades, which are visible in the current frame, have to pass the rendering pipeline. Therefore, different culling methods are presented in Chapter 6.

In Chapter 7 we present the different stages of the rendering process for each blade of grass.

Chapter 8 provides additional information of implementation details.

The results of our approach are shown and discussed in Chapter 9.

Finally, Chapter 10 gives a conclusion and states some future work to further improve the algorithm.

CHAPTER 2

State of the Art

In this chapter we present important rendering techniques in the domain of grass rendering, subdivided into image-based, hybrid and geometrical approaches.

2.1 Image-based approaches

The earliest grass rendering techniques use flat textures mapped onto the ground. Even nowadays, these methods are frequently used, although they are not realistic or react on their environment.

In 1995, Neyret et al. [Ney95] used three-dimensional textures to render complex repetitive geometries such as fur, grass, foliage or forests. The three-dimensional textures represent the geometry as cylinder slices in each texel. The rendering process first uses ray-tracing of the often repeated volumes of single bunches of grass. Then the blades of grass are rendered as trilinear deformations of cylinders according to the traced texels. Using this technique, large meadows and forests can be rendered with high geometrical detail, but the rendering performance is far behind the requirements of real time. In addition, the field's appearance is homogenous, since the three-dimensional texture is simply repeated. An image of this technique can be seen in Figure 2.1a.

The first real improvements to simple texture mapping in real time are introduced at the beginning of the 21^{st} century. Those improvements were realized throughout the application of billboarding techniques. They use semi-transparent quadrilaterals with blades of grass drawn onto them that are placed all over the scene. The quadrilaterals can either turn themselves towards the camera all the time [Wha05], be placed in fixed layers [PC01] or form stars when seen from above [Pel04]. Each of those techniques has their benefits over the others, but all of them have the drawback that their realistic appearance depends to a large extent on the viewing angle. Example renderings of those unanimated billboarding techniques are shown in Figure 2.2.

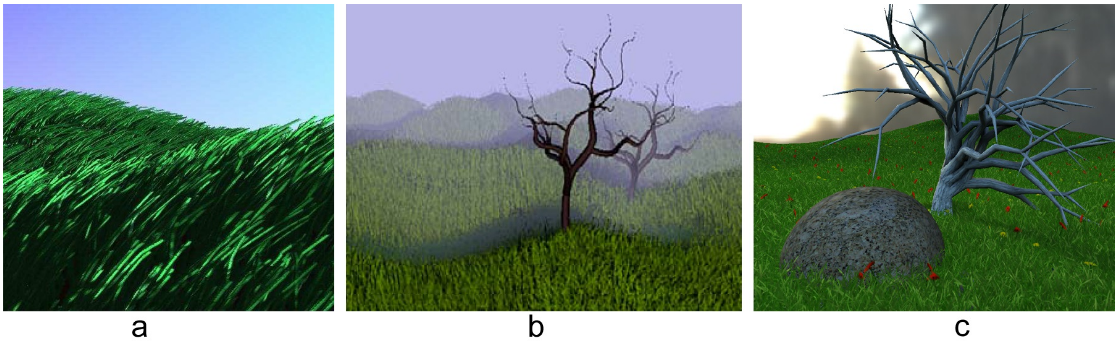

Figure 2.1: This figure shows images of three different image-based grass rendering techniques. a) represents a rendering from the technique proposed by Neyret et al. [Ney95], b) is extracted from the paper by Shah et al. [SKP05] and c) gives an example showing the result of the algorithm of Habel et al. [HWJ07].

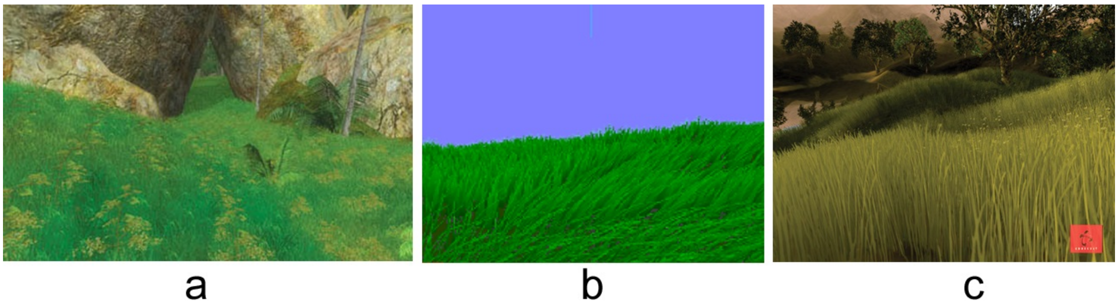

Figure 2.2: This figure shows renderings of three different non-interactive billboard techniques. a) represents an image from the GPU gems article [Wha05], b) is taken from the paper by Perbet et al. [PC01] and c) shows a rendering from the GPU gems article [Pel04].

In 2005, Shah et al. [SKP05] focused again on texture mapping and overcame some of the drawbacks by proposing a method which is animated using bidirectional texture functions. In addition, the authors used displacement mapping at silhouettes to hide the fact that the grass has no height. Although being computationally very expensive, the results still have an obviously artificial appearance. A rendering of this technique can be seen in Figure 2.1b.

Two years later, Habel et al. [HWJ07] menaged to render dense fields of grass with good visual depth. They used half-transparent texture slices being placed parallel or orthogonally to each other. Regarding the rendering, the resultant grid was ray-traced. In addition, the grass can be animated by distorting the texture lookup coordinates according to the wind direction. The major disadvantage of this rendering technique is the dependence between the density of the texture grid and the viewing angle. The steeper the viewing angle, the bigger the amount of needed texture slices to hide the grid structure. An example of this technique is presented in Figure 2.1c.

In 2009, Orthmann et al. [JSK09] proposed a billboard technique consisting of two textures. They are placed orthogonally to each other and have the ability to collide with complex objects. In order to avoid heavy distortions of the texture during collision, a mass-spring model, for example like in cloth rendering, is taken into account for each billboard. On the one hand, the grass is responsive to wind and other forces. On the other hand, the major drawbacks of billboard techniques still exist. Figure 2.3a and b show results of this algorithm.

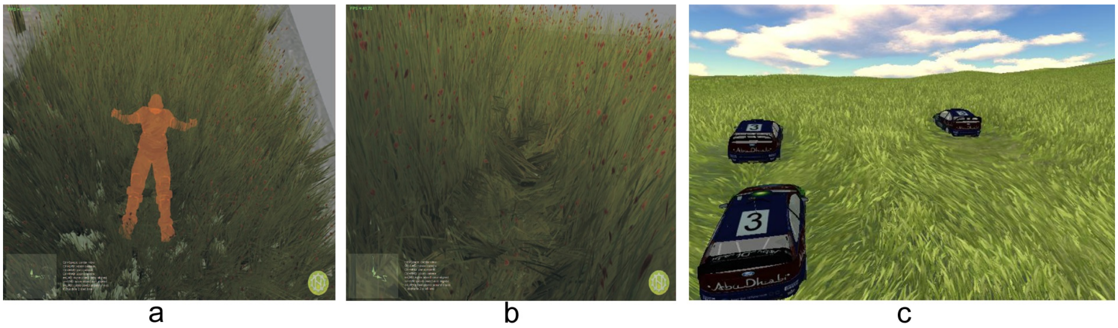

Figure 2.3: This figure shows examples of two interactive billboard techniques. a) and b) are extracted from the paper by Orthmann et al. [JSK09] and c) represents a rendering from the algorithm by Chen et al. [CJ10].

The latest exclusively image-based approach was presented by Chen et al. [CJ10] in 2010. Like Orthmann et al., the authors used a billboard technique as their geometrical representation of a bunch of grass. Although regarding the physical model, they treated the meadow formed by the blades of grass like a wave using dynamics of continuum. In addition, they proposed a procedural approach to handle grass-object and grass-wind interactions, while the grass-grass interaction comes from the dynamics of continuum. A major advantage of this rendering technique is the well coordinated influence of the environment with each bunch of grass. On the other hand, the wavy animation of the meadow does not look very realistic especially at grass-object collisions. An example of this technique can be seen in Figure 2.3c.

2.2 Geometrical approaches

Rendering grass as geometrical object has the advantage that transformations, caused by wind or collisions, can be directly applied to the object and no image distortions have to be done. The first algorithm that only uses a geometrical representation of each blade of grass is proposed by Wang et al. [WWZ$^+$05]. The authors model a single blade of grass with few vertices and draw this object over the whole scene for multiple times. Wind and grass-grass interactions are calculated directly on the graphics card and the displacement result is applied on the skeletal axis of the blade afterwards. A major drawback of this rendering technique is the homogenous appearance of a meadow, since the same object is used for each blade of grass. Figure 2.4a shows a rendering of this algorithm.

In 2009, Zhao et al. [ZLZ09] proposed a rendering technique that draws bunches of grass as geometrical objects using OpenGL's geometry shader. The algorithm is capable of detecting collisions with non-trivial objects directly on the graphics card. Additionally, the bunches of grass can be influenced by wind. The field of grass is very sparse due to performance reasons, although the algorithm uses an additional image-based technique for grass in greater distance. A result of this technique can be seen in Figure 2.4b.

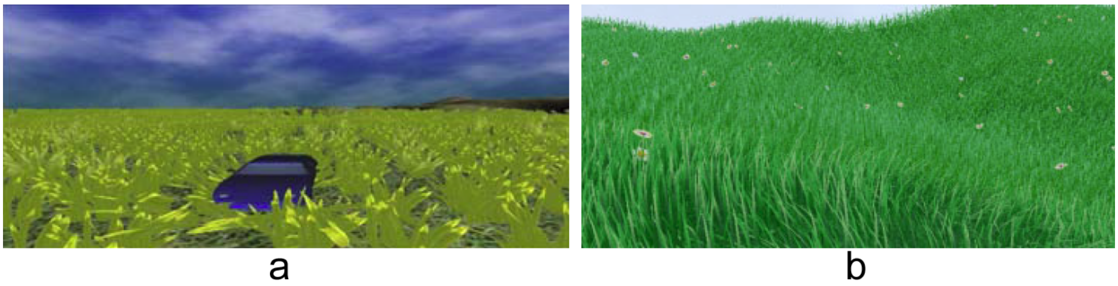

Figure 2.4: This figure shows examples of two geometrical approaches. a) is taken from the paper by Wang et al. [WWZ$^+$05] and b) represents a rendering from the algorithm by Zhao et al. [ZLZ09].

A more novel approach that draws each blade of grass as geometrical object is propsed by Jahrmann et al. [JW13]. The authors use OpenGL's tessellation pipeline to model blades of grass with dynamic level-of-detail directly on the graphics card. A single blade is represented by a tessellated quad, and an alpha texture forms the shape of it. For large fields of grass, a single patch of grass is drawn multiple times, using instanced rendering. In addition, each blade of grass is animated representing a static wind. Collisions are also approximated with a texture indicating the translation. Figure 2.5a shows an example of this technique. The following thesis is built on this algorithm by Jahrmann et al.

The latest approach in the subject of grass rendering was proposed by Fan et al. [FLHS15] in 2015. The algorithm is similar to the work of Jahrmann et al., but the authors turned their attention more on the interactions of each blade of grass with three-dimensional objects. The simulation follows a tile-based pattern and each tile on which an object has been positioned over the period of the last two seconds is updated. After this period of time, the blades of grass of the tile remain in their original shape again. Therefore, the algorithm is capable of rendering and simulating dense fields of grass in real time, but the performance depends to a large extent on the number and size of tiles. A rendering of this technique can be seen in Figure 2.5b.

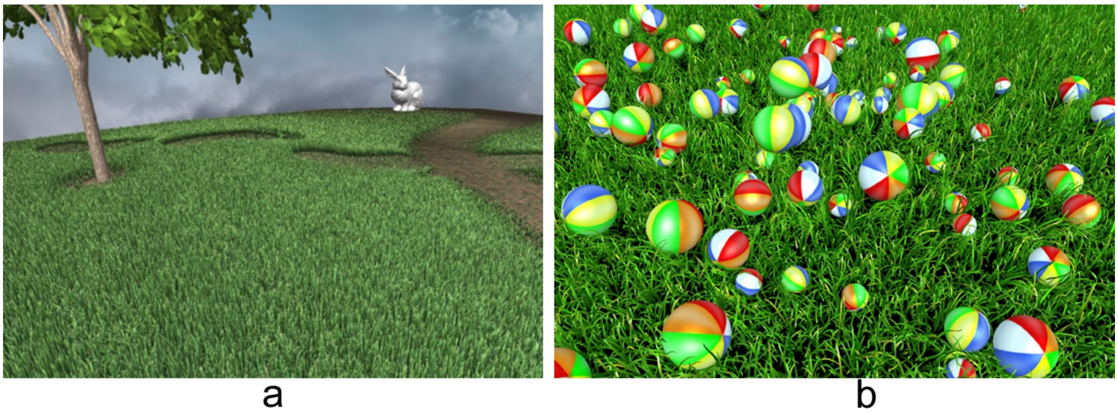

Figure 2.5: This figure shows renderings of the two most modern geometrical approaches. a) represents a rendering taken from the paper by Jahrmann et al. [JW13] and b) shows an image from the algorithm by Fan et al. [FLHS15].

2.3 Hybrid approaches

In 2009, Boulanger et al. [BPB09] combined existing geometrical and image-based approaches and built an algorithm, having different rendering techniques as different static level-of-detail stages. For rendering grass being near to the viewer, the algorithm uses geometrical objects agglomerated to bunches of grass. Some sampled geometrical blades of grass are also rendered as shadow casters. The next level-of-detail stage uses vertical and horizontal texture slices that form a three-dimensional grid, which is similar to the two-dimensional grid by Habel et al. For rendering grass in greater distance, only one horizontal texture slice is taken into account, which corresponds to simple texture mapping. In order to overcome step artifacts of static level-of-detail, the authors interpolated between different levels in order to hide the boundaries. On the one hand, the rendering is very efficient within this technique and has a realistic appearance. On the other hand, the grass representation is not animated and connot interact with its environment. Figure 2.6 shows a visualization of the different static level-of-detail stages and an example of this rendering technique.

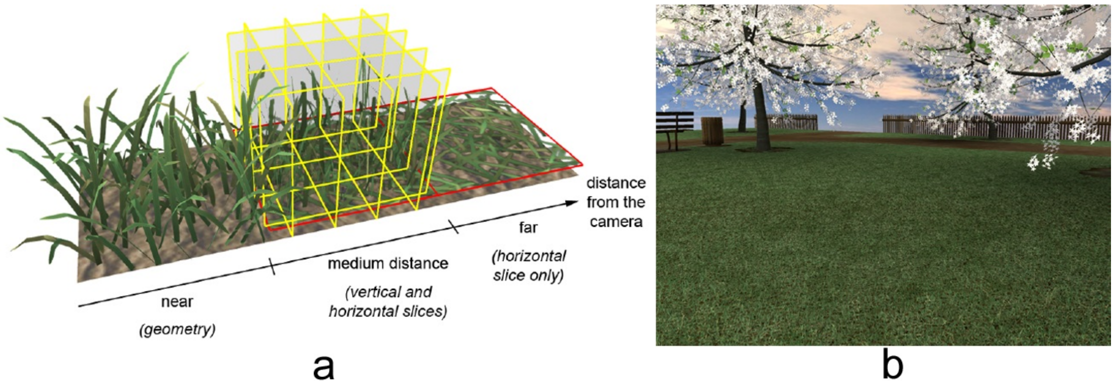

Figure 2.6: This figure shows examples of the hybrid approach by Boulanger et al. [BPB09]. a) represents a visualization of the different static level-of-detail stages and b) shows a rendering from the proposed algorithm. Both figures are extracted from the paper by Boulanger et al. [BPB09].

CHAPTER 3

Overview

The main goal of this work is the generation of a technique that is applicable to render grass in interactive applications, like computer games or simulations. For this reason, the rendering technique has to meet two requirements:

- The rendering technique should be independent of the shape and the orientation of the field of grass. Thus, it should have the ability to render grass on heightmaps and arbitrary three-dimensional objects. Furthermore, each blade of grass should be able to grow in any possible direction.
- The rendering technique should have the capability of evaluating complex physical interactions with each blade of grass while rendering in real time. The physical interactions should include wind and collisions with different objects. These collisions should be computed regardless of the complexity of an object.

In order to fulfill these requirements, each blade has to save its individual state. However, this prevents the usage of related rendering techniques, which are based on drawing the same geometry multiple times. Therefore, we introduce new culling methods, such that only those blades that are visible in the final rendering have to be drawn.

This chapter gives an overview of the structure and the procedure of our grass rendering algorithm. In the following section, the hierarchical model of our concept is shown. Finally, Section 3.2 presents the coarse steps of the algorithm, which are explained in detail in later chapters.

3.1 Conceptional structure

In order to explain the structure of the algorithm, we start with the lowest level of the hierarchy, which is a single *blade of grass* placed on a three-dimensional model. The detailed definition of a blade of grass is shown in the next chapter together with the relevant attributes.

Many single blades of grass are put together to *patches*, which represent the next level of the hierarchy. A patch functions mainly as a container for blades of grass. It has neither a specific shape nor information about the geometrical object on which the blades of grass are placed. It holds the data for all buffers that are needed during the update and the rendering. Furthermore, the patch is responsible for sending each draw call and each compute-shader dispatch to the graphics card.

The next level of the hierarchy primarily exists to reduce the memory footprint and is called *patch-info*. It contains a pointer to a single patch and refers it to a position in the scene. More than one patch-info object is able to point to the same patch, allowing copies of identical blade distributions over the scene. Nevertheless, each positioned patch has its unique state, even if it is just a copy of another patch. The reason is that the influence of forces is saved in a force texture, which refers to each blade of grass on patch-info level instead of patch level. More detailed information on the force texture is given in Chapter 5. In addition, a patch-info object contains the information about the shape of each blade of grass during the rendering. This can also be a useful tool to make the meadow more heterogenous. The shape of a blade of grass will be explained in detail in Chapter 7.

The *field of grass* represents the next level of the hierarchy. It holds the information about positioning, wind effects and gravity, in addition to the list of patch-info objects assigned to this field of grass.

The topmost level of the hierarchy is called *overmind*. It manages all fields of grass together with common rendering attributes like the maximum visible blade distance. In addition, the overmind contains a list of all active collider objects. This list is distributed to the field-of-grass objects for collision detection during the rendering of each frame. Only a single overmind object has to exist for the whole grass-rendering application.

A sketch of the levels of the hierarchy can be seen in Figure 3.1.

3.2 Basic algorithm

Based on the hierarchy presented above, our grass-rendering algorithm can be divided into two general steps. The first step calculates all needed data and prepares the data structures. This step distributes the blades of grass and generates the patches from the single blades. It is only performed once at the start of the application. Detailed information on this preprocessing step is given in Chapter 4. The second step is processed during the rendering of each frame. In this step, three tasks have to be accomplished, which are stated in the list below. Figure 3.2 shows an illustration of the basic algorithm.

1. The physical model is evaluated for each single blade of grass. This includes the computation of collisions with complex objects and the influences of wind and

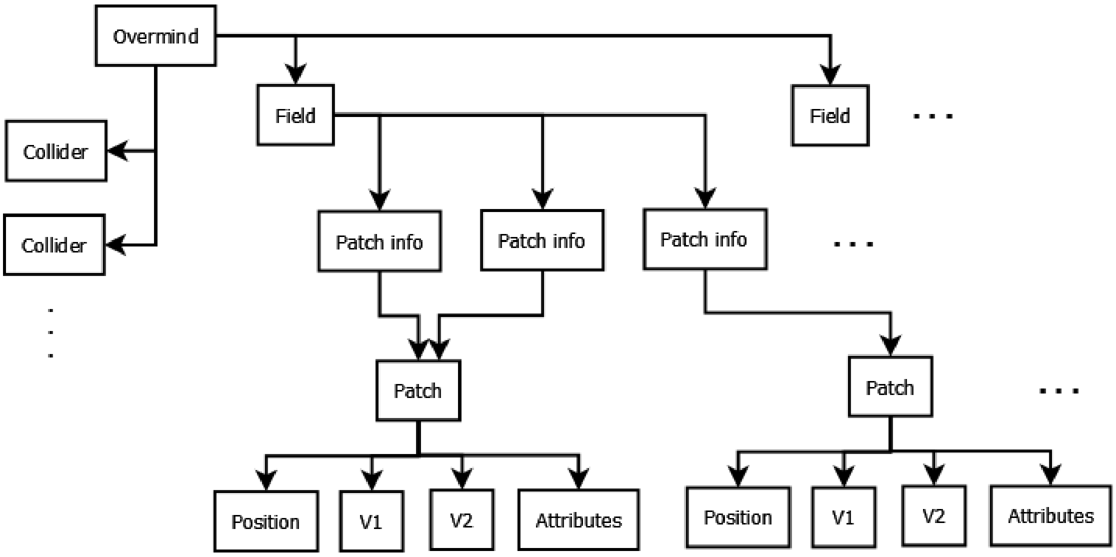

Figure 3.1: This figure shows a sketch of the levels of the hierarchy.

gravity. More detailed information on the physical model is presented in Chapter 5.

2. The visibility of each blade of grass is tested, and a certain fraction of the blades is removed from the rendering pipeline. The decision whether a blade is culled is based on the direction of the blade, its distance to the camera and the occlusion of other objects. This step is crucial for the performance of the proposed rendering technique. Chapter 6 presents the different visibility tests.
3. Each visible blade of grass is rendered as geometrical object. The shape of the blade of grass is defined by an analytic function. The dynamic level-of-detail of hardware tessellation enables smooth edges for the shape. The rendering procedure is explained in Chapter 7.

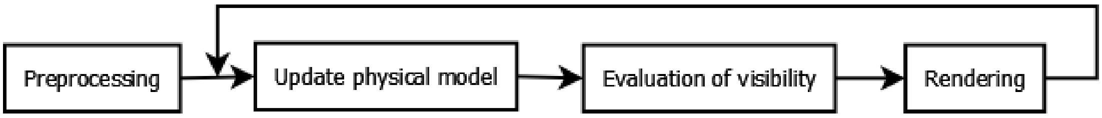

Figure 3.2: This figure illustrates the steps of the basic algorithm.

CHAPTER 4

Preprocessing

In this chapter, we describe the initialization process of a field of grass. The first section presents our definition of a single blade of grass together with its attributes, which is essential for describing the initialization process. The first step of the initialization process is the generation of blades of grass, distributed on top of a three-dimensional model. This generation process is described in Section 4.2. When the generation of the blades of grass is finished, the blades are divided into patches. Section 4.3 deals with the generation process of the patches. Additional information on implementation details of the patch-generation algorithms is presented in Chapter 8.

An important contribution of our grass-rendering technique is that the blades of grass can be distributed on arbitrary three-dimensional models. Thus, the distribution is not bound to flat surfaces or heightmaps. This is in clear contrast to the related rendering algorithms by Jahrmann et al. [JW13] or Fan et al. [FLHS15], for example. In addition, the patch-generation algorithms have to be capable of operating without knowledge of the three-dimensional model beneath the blades of grass. This enables any kind of model to be used as basis for the grass rendering technique.

Regarding the three-dimensional model, the algorithms in this thesis are specialized for meshes consisting of triangles. This is no limitation, since triangles are the standard representation used in the rendering pipeline.

4.1 Definition of a blade of grass

In this section, we define the model of a single blade of grass and state the important attributes, which are essential for the understanding of further sections. Figure 4.1 presents a sketch of the definition of a blade of grass. Like in our previous paper [JW13], each blade is defined by a quadratic Bézier curve. Therefore, each blade needs to save the coordinates of the three control points. The first and the third point reference the blade's *position* on the ground and its tip, $\mathbf{v}_2$. The second control point, $\mathbf{v}_1$, indicates the bending of the curve.

Apart from the curve representation, each blade of grass is defined by four attributes. Two attributes decide the size of the blade by expressing its *width* and *height*. Furthermore, the flexibility of a blade of grass is defined by the third attribute *bending*, which is a value between zero and one. The fourth attribute describes the *spatial alignment*, which consists of a three-dimensional vector and a single value. The three-dimensional vector indicates the growth direction of the blade of grass. This vector is named *up-vector* in the rest of the thesis. Additionally, the value in the intervall of $[0, 2\pi]$ expresses the blade's direction on the plane defined by the up-vector.

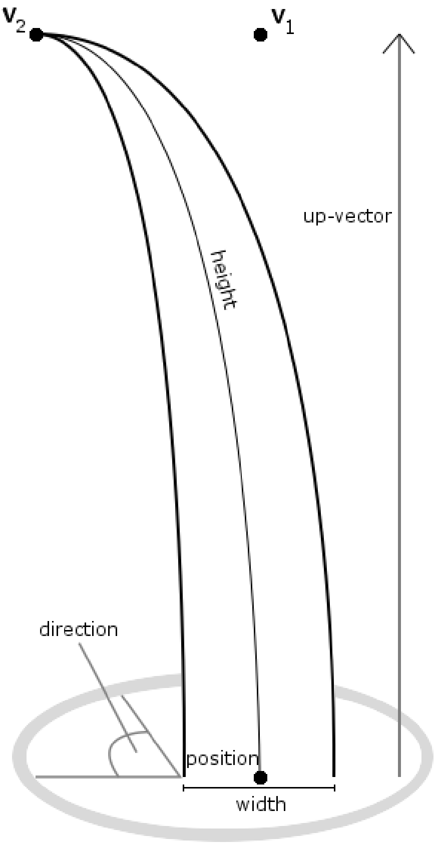

Figure 4.1: This figure outlines the definition of a blade of grass. The Bézier curve is shown together with its three control points. Additionally, all attributes apart from the bending value are presented together with their impact on the blade of grass.

4.2 Distribution

The following sections explain the generation process of the blades of grass on the surface of a three-dimensional model. This process consists of two steps, which are iterated until the desired amount of blades are generated. First, a point on the surface of the three-dimensional model is found. Afterwards, either a single blade or a tuft of grass is seeded at the selected position. In order to ensure that tufts of grass are not clumped together, we use a Poisson-disk sampling approach [Mit87]. For single-blade seeding, we do not use Poisson-disk sampling, since random clumping of individual blades is beneficial for the natural appearance of the field of grass. The scheme of the generation process is shown in Figure 4.2.

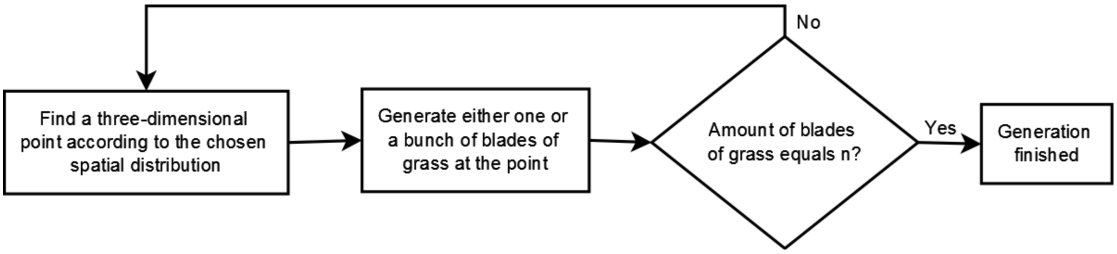
No
Find a three-dimensional point according to the chosen spatial distribution
Generate either one or a bunch of blades of grass at the point
Amount of blades of grass equals n?
Yes
Generation finished

Figure 4.2: This figure presents the scheme of the algorithm for distributing blades of grass on the surface of a three-dimensional model.

The blades are generated on the surface of a three-dimensional model, which is required to be represented as a triangle mesh. The amount of blades that are generated during this process is calculated with respect to the area of the surface of the three-dimensional model. This calculation is performed by multiplying the area of the surface with a given *density* parameter, which indicates the amount of blades on one square unit. Equation 4.1 shows this calculation, where n represents the amount of blades, A_i is the area of the face i and d refers to the density parameter.

$$n = d \sum A_i \tag{4.1}$$

For the first step of the generation algorithm, random points on the triangle mesh have to be found, which represents the spatial distribution of the blades of grass. The following section presents different algorithms for finding points on the surface of a mesh. In the second step, the blades are seeded by either generating a single blade or a whole tuft of grass for each point selected by the first step. This is described in Section 4.2.3. Between these two steps, the Poisson-disk sampling can reject points given by the spatial distribution before the seeding process is executed. This additional sampling is described in Section 4.2.2.

4.2.1 Spatial distribution

In the following, we describe three algorithms for finding a random point on the surface of a three-dimensional model. After a point is selected, the seeding process is executed to generate either a single blade or a tuft of grass. If the number of generated blades is less than n or if the selected point is rejected by the Poisson-disk sampling of the tuft seeding, the spatial distribution computes the next random surface point.

Random-face distribution

As one of the three algorithms, the random-face distribution generates a three-dimensional point on the surface of a model. It randomly selects a face of the model and calculates a random point inside this triangle. Each point of a triangle can be expressed by using barycentric coordinates. These three-dimensional coordinates can also be used for the interpolation between the vertices of a triangle. Thus, a point represented by barycentric coordinates is inside the triangle if each barycentric coordinate is positive and the sum of the coordinates does not exceed one. Equation 4.2 shows the computation of a random point inside a triangle using barycentric coordinates. In this equation, $\mathbf{b}$ refers to the barycentric coordinates and r_1, r_2 and r_3 are random values. Furthermore, $\mathbf{P}_B$ indicates the interpolated position and P_i is the i^{th} vertex of the triangle.

$$\begin{aligned} \mathbf{b} &= \frac{[r_1, r_2, r_3]}{r_1 + r_2 + r_3} \\ \mathbf{P}_B &= \sum_{i=0}^{3} b_i * \mathbf{P}_i \end{aligned} \tag{4.2}$$

The result of the random-face distribution is shown in the left column of Figure 4.3.

Area distribution

In contrast to the random-face distribution, the area distribution tries to populate each face of the model with the same point density. To do so, it calculates the capacity of generated points of each face according to its area and the given density. In addition, during the whole process, each face counts the number of points that are generated inside it. This number must not exceed its capacity, as long as another face has room for additional points. The calculation of the capacity is shown in Equation 4.3, where n_i respectively A_i refers to the capacity and area of the i^{th} face. In addition, d is the density parameter.

$$n_i = \lfloor d \; A_i \rfloor \tag{4.3}$$

During the generation process, the area distribution computes a random point on a face, starting from any face of the model. For the next iterations, the same face is used for

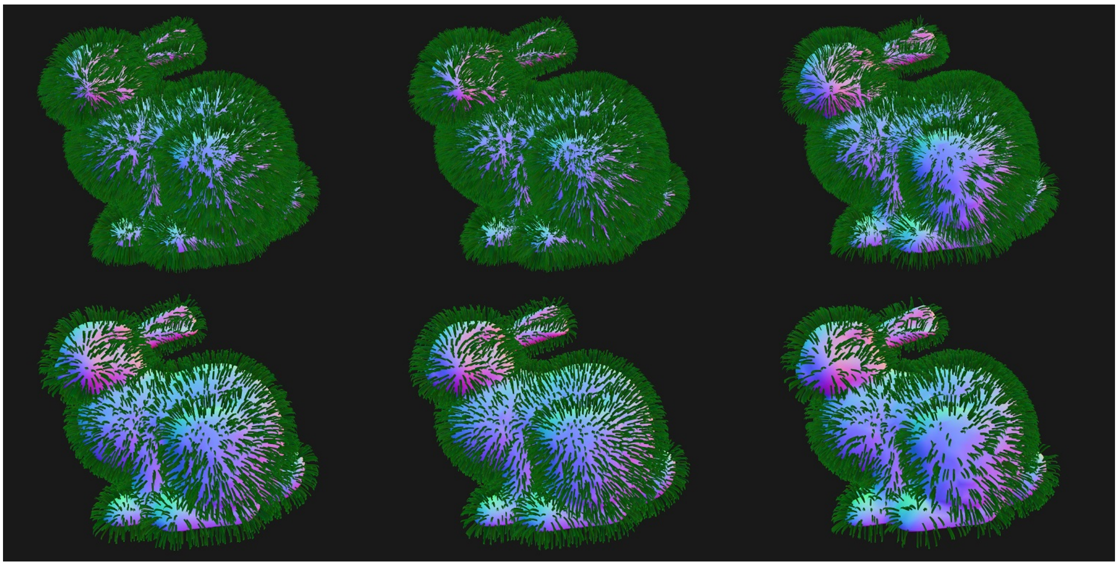

Figure 4.3: This figure presents the comparison of different spatial distribution in combination with different blade generation algorithms. The rows represent the blade generation algorithms: the upper row refers to the single-blade seeding and the lower row to the tuft seeding. The columns represent the spatial distribution algorithms: the left column refers to the random-face distribution, the middle column to the area distribution and the right column to the octree distribution. The *bend* parameter of the blades of grass is set very low, such that the positioning of the blades is more visible. The three-dimensional model is taken from the Standford scanning repository [Lab].

the generation of a random point. If the face has distributed all points according to its capacity, the next face is selected. This procedure continues until all faces have filled their capacities.

However, since each face produces only as many points as its area indicates, there are fewer points than desired due to the rounding. Thus, the rest of the points are generated randomly on the faces that do not have blades assigned. If there are no faces without blades, the rest of the blades are generated at any face at random. The scheme of the area distribution can be seen in Algorithm 4.1. Two examples of the area distribution are presented in the middle column of Figure 4.3.

Volumetric distribution

In contrast to the distributions that are mentioned above, the volumetric distribution aims at generating points in uniform density over the three-dimensional space. This can be useful for terrains that have a high variaton of the curvature. If all faces are populated with the same density, many points are generated in narrow gaps although the space is not suited for many blades of grass. This case is illustrated in Figure 4.4, where the upper image shows the area distribution and the lower image represents the volumetric

Algorithm 4.1: Distribution - Face area

```
points ← 0;
for each face $f$ do
    $N_p$ ← $f$.area $*$ density;
    if $N_p \geq 1$ then
        generate $\lfloor N_p \rfloor$ points;
        points ← points + $\lfloor N_p \rfloor$;
    end
end
if points $< n$ then
    generate rest on faces without points or random if there are no faces without points;
end
```

distribution. In addition, the red lines of this figure illustrate the generated blades of grass and the green dots the distribution of random samples. During each iteration of the generation process, the volumetric distribution selects a random point inside the bounding box of the three-dimensional model and searches for the nearest surface point. In order to be able to search for the nearest surface point in reasonable time, we use an octree data structure to accelerate this process. A pseudocode implementation can be seen in Algorithm 4.2.

First of all, this distribution builds an octree of the model's vertices and saves the corresponding faces at each vertex. Afterwards, a point in the bounding box of the model is chosen randomly and the nearest vertices are found, using the octree data structure. This gives us a list of candidate faces, on which we find the closest point with respect to the random point. Afterwards, the distance to each of those closest points is calculated and the nearest one is selected. The result can be seen in the right column of Figure 4.3.

When analysing the algorithm, it can be easily noticed that the density is not equal over the model. Faces in regions with large empty volume, for example, are favored in the seeding process. This is no drawback of the volumetric distribution, since it aims at a uniform distribution over the three-dimensional space rather than over the surface of the model. In the right column of Figure 4.3, this occurance of a higher density is visible at the back and the chest of the bunny, whereas the nose, ears and feet have less density.

4.2.2 Poisson-disk sampling

In order to ensure that the points from the spatial distribution are not randomly clumped together, we introduce a Poisson-disk sampling approach that filters out points being too close to previously distributed points. Our approach follows the principle of the dart-throwing algorithm [Mit87]. The dart-throwing algorithm takes a given minimum distance d_{min} and evaluates the distances to all samples for each new random sample

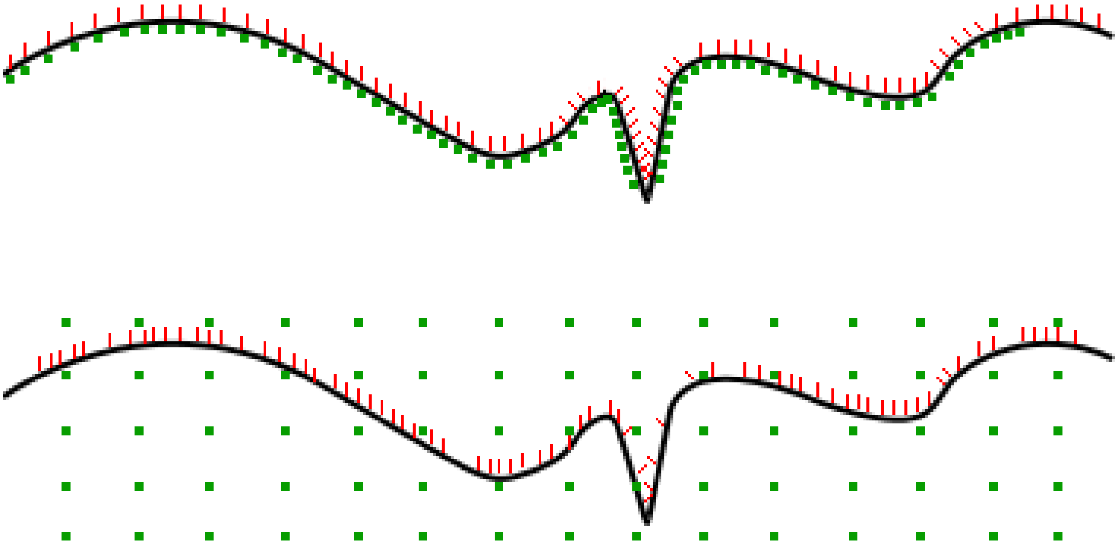

Figure 4.4: This figure compares the distribution betwen the area distribution (upper image) and the volumetric distribution (lower image) for a special case. The red lines indicate the generated blades of grass and the green dots represent the randomly sampled points of each distribution. Both images contains the same amount of blades and sample points.

point. If another point is located within the distance d_{min}, the sample is rejected.

In our case, the initial value for d_{min} is given by the seeding algorithm. However, we decrease d_{min} steadily over the whole generation process, which has two advantages. On the one hand, the distribution of the field of grass becomes more heterogenous, since early samples are more sparse than later samples. On the other hand, the Poisson-disk sampling needs fewer iterations to be calculated. Therefore, if the Poisson-disk sampling is enabled, the random point from the spatial distribution is tested whether its distance to a previously generated points is less than d_{min}. If that is the case, the point is rejected and the spatial distribution is executed instead of the seeding process. After a certain amount of surface points are rejected, the minimum distance d_{min} becomes lower the more samples are rejected.

4.2.3 Seeding

After a point is selected by the spatial distribution, the seeding process is executed. There are two possibilities how blades of grass can be seeded based on a surface point, which are presented in the following sections. Either a single blade or a tuft of grass is generated. In order to ensure that tufts of grass are not clumped together, we enable the Poisson-disk sampling approach, which is explained in the previous section, for the tuft-seeding process. In case of seeding single blades of grass, the random clumping of single blades of grass

Algorithm 4.2: Distribution - Octree

```
build octree $O$ with the vertices of the 3D model;
for $i$ from 0 to n-1 do
    generate random point $P$ inside the model's bounding box;
    List of faces $F_i$ is retrieved from $O$.getNearestVertices($P$);
    $P_{nearest} \leftarrow 0$;
    for each $F_i$ do
        $P_F \leftarrow$ closest point from $P$ onto $F_i$;
        if $P_F \in F_i \wedge distance(P, P_F) is\ minimal$ then
            $P_{nearest} \leftarrow P_F$;
        end
    end
    select point $P_{nearest}$;
end
```

is desired to provide a natural appearance of the field of grass. Therefore, we do not enable the Poisson-disk sampling approach for the single-blade seeding. After the desired number of blades are generated, the number of already generated blades is evaluated. If this number has not reached n, the spatial distribution is processed again to find a new random point on the surface.

As shown in Figure 4.5, the single-blade seeding is better suited for covering large fields of grass. On the other hand, the tuft seeding better resembles a meadow from the real world, since grass grows in bunches naturally.

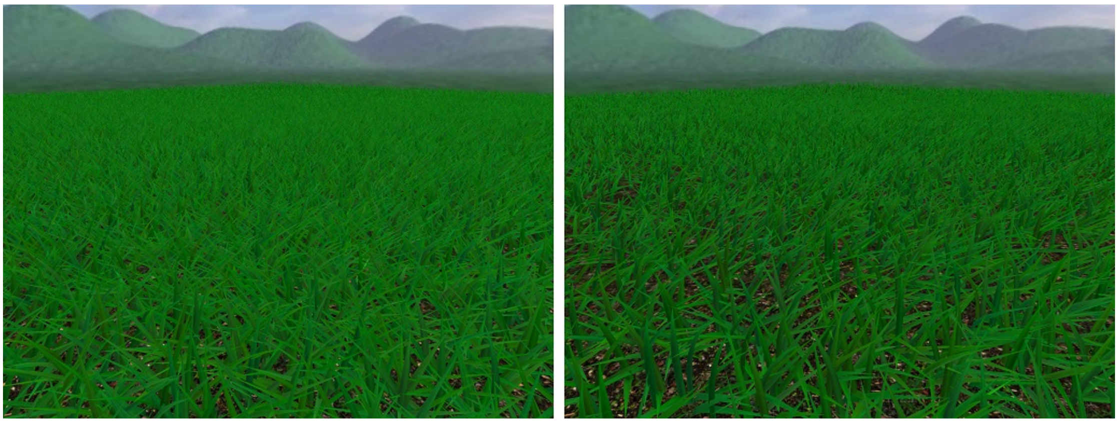

Figure 4.5: This figure compares the two different possibilities of blade generation. The left image shows the single-blade seeding and the right image the tuft seeding. The fields of grass of both images have the same amount of blades.

Single-blade seeding

The single-blade distribution generates exactly one blade at the position of the input surface point. The attributes – width, length and bending – of the blade are chosen with respect to the minimal and maximal values given as parameters. The blade's up-vector is defined by the normal vector of the surface at the specific point and the direction value is chosen randomly from the interval $[0, 2\pi]$. The result of the single-blade distribution is shown in the upper row in Figure 4.3.

Tuft seeding

Instead of a single blade, the tuft-seeding process generates multiple blades of grass for the input surface point. For this, the amount of tufts m needed to generate n blades is calculated using Equation 4.4. In this equation, b_{mean} refers to the average amount of blades of grass that are generated for each cluster. This value is given as parameter.

$$m = \left\lceil \frac{n}{b_{mean}} \right\rceil \tag{4.4}$$

In addition to the generation of blades, the Poisson-disk sampling, which is described in Section 4.2.2, is enabled and ensures that the tufts are not clumped together. As input for the Poisson-disk sampling, an initial value for the minimum distance, d_{min}, between the tufts of grass has to be calculated. For this calculation, we imagine a rectangle having an area according to the area of the surface of the three-dimensional model. With this we choose the value of d_{min} by dividing the area by m. This calculation is presented in Equation 4.5, where A_i refers to the area of the i^{th} face and m indicates the amount of tufts.

$$d_{min} = \sqrt{\frac{\sum A_i}{m}} \tag{4.5}$$

For the imagined rectangle, this assignment of d_{min} would lead to a perfect disk packing, which provides a sparse distribution for early tufts of grass. When a surface point is accepted by the Poisson-disk sampling, the blades of grass are generated. In order to have a better variation between the different tufts, a small variance is added to the given average amount of blades for a tuft to calculate the amount of blades, b, that have to be generated. This variance is defined by a maximum of twenty percent of b_{mean}. Thus, the amount of blades that have to be generated can be derived using the following equation. In this equation, r represents a uniformly distributed random number, n is the amount of blades to be generated and n_c is the number of already generated blades.

$$b = \min\left(b_{mean} \pm r\frac{b_{mean}}{5}, n - n_c\right) \tag{4.6}$$

The last part inside the minimum calculation of the equation above ensures that the desired number of blades, n, cannot be exceeded, even if b is modified by some variance.

In order to create a realistic field of grass, the blades within a tuft of grass should have a similar appearance, whereas the appearance of blades of different tufts should be more variant. This is achieved by defining new intervals for a blade's attributes for each tuft. Certainly, the new intervals have to be located inside the intervals given as parameters. The attributes of the blades of a tuft of grass are calculated in a pseudo-random way. The pseudo-random generation is motivated by the generation of a patch of blades of grass in the algorithm of Boulanger et al. [BPB09]. Thus, when generating b blades of grass, the attribute values are equally distributed over the tuft's intervals. Considering the case of generating three blades of grass, the attributes of one blade get the maximal values, the attributes of the second blade get the mean values and the attributes of the third blade get the minimal values.

In addition to the attributes, the direction of the generated blades is also equally distributed over the interval $\left[0, \frac{2\pi}{b}\right]$ and the up-vector of each generated blade is set according to the normal vector of the surface at the center position of the tuft of grass. For computing the positions of the generated blades of grass, a circle, which radius is defined by a parameter, is generated around the position of the tuft on the local plane defined by the up-vector. Each blade of grass of a tuft is translated inside this circle, where the direction value of the respective blade is used as angle that defines a vector in a circle.

The result of the tuft seeding is presented in the lower row in Figure 4.3.

4.3 Patch generation

Representing each single blade as individual geometric object would create an enormous amount of objects on the graphics card and would definitely exceed the memory limits of the hardware due to the large amount of generated buffers. Therefore, we group the blades of grass into patches, which refer to the geometrical representation of the field of grass on the graphics card. A patch is defined by the blades of grass that are assigned to it. Thus, the geometry of a patch can have any spatial extent. In addition, each patch stores its bounding box, which is later used for visibility tests. An important feature of our patch-generation algorithms is that they are capable of processing fields of grass of arbitrary three-dimensional shape without having knowledge of the underlying model. In order to achieve maximum performance with a minimum amount of used memory, a patch-generation algorithm has to accomplish three main goals as well as possible:

- The amount of blades of each patch should be equal. This enables a fast execution of compute shader tasks, which is necessary for several steps of our technique.

- The amount of blades of each patch should be approximately a square of a natural number. This minimizes the memory footprint of our technique, because the data can be more tightly packed.
- The spatial extent of a patch should be compact and rectangular. This minimizes the size of the bounding box of the patch.

Thus, we need to generate m patches, where each patch contains n_p blades of grass. The number of patches is important for the performance of the algorithm. If too many patches are generated, the graphics card is busier with switching data than with processing it. On the contrary, the bounding box of each patch is larger if less patches are generated. Therefore, the amount of patches has to be adapted to the limitations of the graphics card. In our case, the maximal number of threads of a compute shader dispatch is important, since many steps of the algorithms are implemented using compute shaders. This maximizes the workload for a single dispatch and reduces the number of dispatches. Since the optimal number depends on the graphics hardware, we can only give an idea on how the optimal number is calculated. This is shown in Equation 4.7, where m represents the amount of patches, D_{max} is the maximal number of threads of a compute shader dispatch and f is an empirical factor that depends on the graphics card. In our application, we set f to 10, which achieved the best results for an NVIDIA GeForce GTX 780M graphics card.

$$m = \left\lceil \frac{n}{D_{max} \cdot f} \right\rceil \tag{4.7}$$

An intuitive approach for patch generation would use a grid, where the blades falling into grid cell form a patch. However, this approach would fail in two ways. On the one hand, if a uniform grid is used, the density of the blades of grass has to be perfectly even over the whole field of grass in order to ensure that all patches have an equal amount of blades. On the other hand, when the grass is distributed on top of a complex model, the initialization of a suitable grid would be a hard optimization problem itself.

In the following, we represent patch generation as a clustering problem. Clustering problems try to find groups inside data, where an element is similar to other elements of the same group, but different to elements of other groups [Ber06]. These groups are called clusters, which in our case represent the patches, and similarity in our case is geometric closeness. Since we aim at finding patches of an equal number blades of grass, the clustering problem is constrained to clusters of equal amount. This is called a balanced clustering problem [MF14], which can be solved efficiently using linear programming or graph structures. In the following section, we propose an accurate optimization algorithm, which solves the balanced clustering problem. However, this algorithm needs much time for the calculation. Therefore, we also introduce some ad-hoc algorithms, which produces less accurate results. In spite of the less accurate results, these algorithms are perfect for testing, since they can produce fairly compact clusters very fast.

Some illustrations of the resulting patch divisions of the different methods are presented in Chapter 9.

4.3.1 Clustering algorithm

In this section, we propose an algorithm that generates patches as a result of solving a balanced clustering problem. Balanced clustering is a subcategory of the clustering problem, which adds the additional constraint that all clusters have to have an equal amount of elements. Malinen et al. [MF14] propose a balanced k-means algorithm, which adapts the original k-means algorithm [Har75] to be applicable for the balanced clustering problem. For this clustering approach, the authors use the Hungarian algorithm [BDM09], which operates on graphs structures, to solve the assignment problem of assigning n points to m clusters having n/m slots.

In our application, we implemented a simpler (but slower) approach to solve the balanced clustering problem, which does not need any graph structure or linear program. Initially, a mapping between the blades of grass and the m clusters has to be found. Each of the ad-hoc algorithms, which are described in the following sections, can be applied to provide an initial mapping. However, in order to have a well-functioning optimization step, the initial clusters should not be disjointed for the most parts. Therefore, the nearest neighbor algorithm, which will be explained in Section 4.3.2, provides initial clusters well-suited for the following optimization task.

After the initial clusters are found, the optimization step is performed. The idea of this optimization derives from Stevenson [Ste13], who implemented a balanced k-Means algorithm for the two-dimensional case. In our implementation, the mean of each cluster is calculated first. In addition, each cluster finds the blades that are both nearest to each other cluster's mean and farthest to the own cluster's mean. These blades of grass are called swap candidates. After the swap candidates are computed, all clusters check with each other how they change if the respective swap candidates would be assigned the other way around. When the distances from the differently assigned points to the cluster means are shorter than the distances of the original assignment, the new assignment is taken into account. This optimization iteration, including the recalculation of the cluster means and the swap candidates, is repeated until no cluster swaps any blade with another cluster anymore. Finally, the patches are formed by the blades of grass of the respective cluster. The scheme of the optimization process can also be seen in Algorithm 4.3, where the recalculation of the mean and the swap candidates is only done if C_i or C_j has been swapped in the last iteration.

Depending on the initial clustering, the optimization might stay in a local optimum, where the clusters are not perfectly compact and have a less rectangular shape than possible. For large fields of grass, this optimization procedure can last for hours. Therefore, this expensive algorithm should only happen once and the results should be saved for later uses.

Algorithm 4.3: Cluster optimization algorithm

```
Data: Blades of grass $B$, cluster assignment $C$, swap candidates $S$
swapped ← true;
while swapped == true do
    swapped ← false;
    for each cluster $C_i$ do
        calculate cluster mean $M_i$;
    end
    for each cluster $C_i$ do
        for each cluster mean $M_j$ with $i \neq j$ do
            for each blade $B_k \in C_i$ do
                candidate value ← $distance(B_k, M_j) - distance(B_k, M_i)$;
            end
            $S_{ij}$ ← blade $\in C_i$ with highest candidate value;
        end
    end
    for each cluster $C_i$ do
        for each cluster $C_j$ with $i \neq j$ do
            if $distance(S_{ij}, M_j) + distance(S_{ji}, M_i) <$
              $distance(S_{ij}, M_i) + distance(S_{ji}, M_j)$ then
                $S_{ij}$ moves to $C_j$;
                $S_{ji}$ moves to $C_i$;
                swapped ← true;
            end
        end
    end
end
```

4.3.2 Nearest-neighbor algorithm

The nearest-neighbor algorithm is an ad-hoc algorithm that tries to cluster the blades of grass using a greedy approach. First of all, the largest dimension of the field of grass is evaluated and the blades of grass are sorted according to the coordinate value of the chosen dimension. In the following, each blade that is not yet assigned to a cluster forms a new cluster with the n_p nearest unassigned blades of grass, beginning from the first blade on. Due to the initial sorting, this leads to clusters having the shape of scales. Finally, the patches are formed by the blades of grass of the respective cluster. A more detailed view of this technique is shown in Algorithm 4.4. The advantage of this algorithm is that the patches can be computed fast. In addition, the resulting patches are rarely disjointed.

Algorithm 4.4: Nearest-neighbor algorithm

```
Data: Blades of grass $B$, patch assignment $P$
sort blades along the largest dimension;
$i \leftarrow 0$;
for each blade $B_j$ do
    if $B_j \notin P$ then
        add $B_j$ to $P_i$;
        $N \leftarrow n - 1$ nearest blades;
        for each neighbor $N_k$ do
            add $N_k$ to $P_i$;
        end
        $i \leftarrow i + 1$;
    end
end
```

4.3.3 Largest-dimension sorting

The largest-dimension sorting algorithm is the most simple patch-generation algorithm, which represents the patch generation process as a spatial subdivision. In the same ways as the nearest-neighbor algorithm, which is stated above, it sorts the blades along the largest dimension. Afterwards, each of the m patches is formed by n_p consecutive blades of grass. The major advantages of this technique are that the patches are generated fast and have a rectangular shape. However, the patches are not compact, apart from the special case when a field of grass has a major extent in only one dimension.

4.3.4 Seed-point algorithm

The seed-point algorithm tries to solve the patch-generation problem in two steps. First, it follows a spatial subdivision approach and then it performs a nearest-neighbor clustering on the result of the spatial subdivision.

The idea of the spatial subdivision is the seeding of m uniformly distributed points inside the bounding box of the field of grass. The problem of finding suitable seed points is that the algorithm has to be capable of processing a volume of arbitrary size to find an arbitrary number of equally distributed points. In the following, we explain two simple approaches, which generate seed-points inside the bounding box of the field of grass.

- Box subdivision: This algorithm uses three-dimensional boxes defined by minimum and maximum coordinates. The algorithm is a simple version of the spatial subdivision of a kd-tree [Ben75], where each splitting plane is placed in the center of the largest dimension of the box. In the beginning of the algorithm, there is only the bounding box of the field of grass. During m iterations, the box, respectively a subbox for later iterations, is subdivided along its longest dimension, which increases the amount of boxes by one after each iteration. After the iterations, there are m boxes. Finally, the m seed points are generated at the centers of the boxes. This procedure works best if m is equal to a power of two.

- Dimension factorization: This method takes the number m and divides it into all possible variations of three factors, similar to prime decomposition. The procedure of the factorization is shown in detail in Algorithm 4.5. When all possible factorizations are found, the seed points are generated by searching the best factorization for the dimensions of the bounding box of the field of gras. In order to achieve this result, the ratios of the three dimensions of the bounding box are calculated. The same is done for each factor triple. The triple whose ratios match the volume's ratios best is selected. Afterwards, each dimension of the volume is divided according to a factor of the triple. In order to retrieve the final seed points, the center of each subdivision of the volume is computed. This procedure works best if m can be divided into many different factorizations. If m is a prime number, the m seed points are generated in a line along the largest dimension.

After the seed points have been generated, the nearest-neighbor clustering is performed. Each seed point, beginning from the first one that is generated, searches its nearest n_p blades of grass and generates a patch out of these blades. The advantages of the seed-point algorithm are the fast generation of the patches and the robustness regarding the size and shape of the bounding box of the field of grass. However, the ordering of the seed points is crucial, and later-generated patches are often distorted, since they are formed by the leftover blades of grass.

4.3.5 Advanced seed-point algorithm

A major drawback of the seed-point algorithm is that blades that are located at the border of the bounding box of the field of grass are not prioritized specially and therefore often end up being in one of the last patches together with other leftover blades of grass. This is a problem because disjonted patches have large bounding boxes, which decreases the performance of our algorithm. Thus, the advanced seed-point algorithm is an improvement to the simple form of the algorithm stated above, since it prioritizes blades of grass that are placed far away from all seed points during the clustering.

The procedures of finding the seed points are still valid for this variation, but within this method the clustering is done in a different way. Instead of a simple nearest-neighbor clustering, each blade calculates the sum of distances to each seed point. With this result, a biased distance value v_i for the i^{th} cluster is calculated using the following formula, where d_j is the distance from the position of a blade to the seed point j respectively d_i the distance to the seed point i.

$$v_i = \sum_{j=0}^{m} d_j - d_i \tag{4.8}$$

Algorithm 4.5: Factorize 3

```
Input: $m$ - number of seed points to generate
Output: $T$ - list of factor triples $T_i$
$i \leftarrow 0$;
$m_{max} \leftarrow \lceil \sqrt[3]{m} \rceil$;
for $j = 1$ to $m_{max}$ do
    if $n$ mod $j == 0$ then
        $l \leftarrow \frac{n}{j}$;
        $l_{max} \leftarrow \lceil \sqrt{l} \rceil$;
        for $k = j$ to $l_{max}$ do
            if $l$ mod $k == 0$ then
                $T_i \leftarrow \left\{ j, k, \frac{l}{k} \right\}$;
                sort elements of $T_i$;
                $i \leftarrow i + 1$;
            end
        end
    end
end
```

After all calculations are finished, the seed points cluster the blades of grass according to the biased distance values. Each seed point, beginning from the first one that is generated, searches the n_p blades of grass that have the largest biased distance values and generates a patch out of these blades. Since this clustering is more biased to blades on the border of the bounding box of the field of grass, the results are much better than those from the simple seed point algorithms due to more compact patches. However, the drawback of the distorted and disconnected later generated patches still exists.

CHAPTER 5

Physical Behavior

In this chapter we introduce the physical model of our grass-animation system together with the used data structures. During the computation of each frame, the physical model is evaluated for every single blade of grass. The entire computation is executed on the graphics card using compute shaders. In order to be independent of the frame rate of the application, the calculations of the physical model are multiplied with the time span of the last frame Δt.

The physical model is the most important feature of our grass-rendering technique. In comparison to the work of Jahrmann et al. [JW13], the movement of a blade of grass is not built as a simple translation. Instead, the movement follows the rules of the physical model. Amongst others, this ensures that the length of each blade of grass stays consistent regardless of the influencing forces. In addition, all influences on a blade of grass are put into a single physical model. This is in contrast to the algorithm of Fan et al. [FLHS15], where the collision and the wind calculations are performed independently of each other.

In the first section, we present the force map as a data structure to store the forces influencing a blade of grass. In the later sections, the different influences are shown. The order of the sections represents the order of their calculation. The forces can be divided into natural and collision forces. In order to have accurate collision results, the model has to be evaluated for the natural forces first. Therefore, the position of the control points of a blade of grass $\mathbf{v}_1$ and $\mathbf{v}_2$ have to be computed both before and after the calculation of the collision forces. To compute the position of the control points, three calculations have to be done:

- Ground collision (see Section 5.5)
- Calculation of $\mathbf{v}_1$ (see Section 5.6)

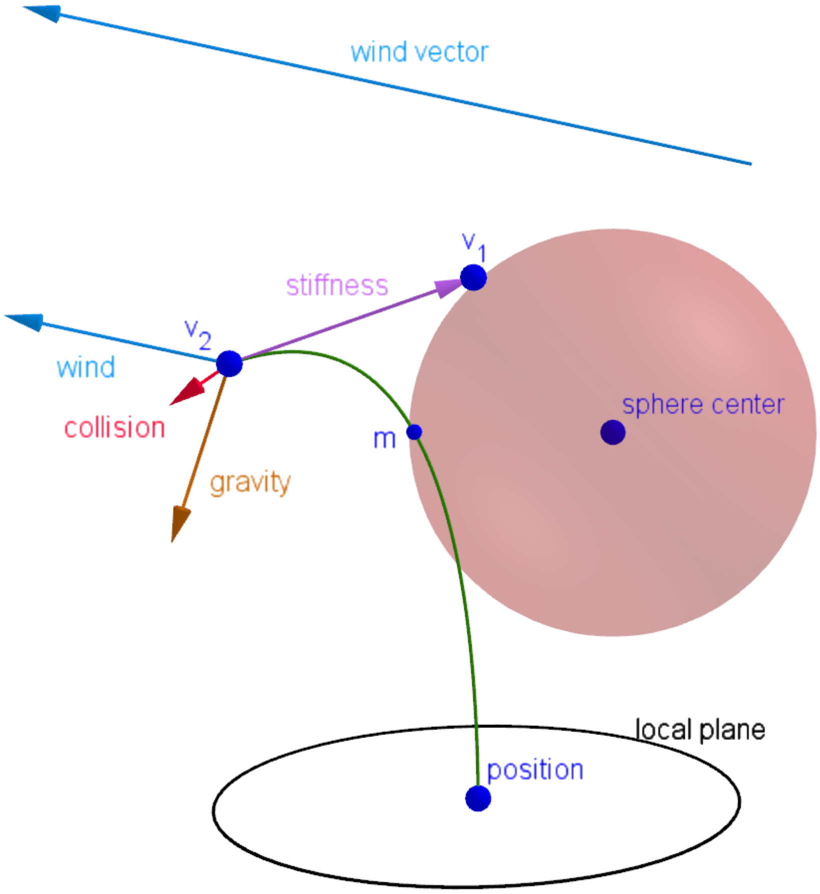

Figure 5.1: This figure shows an illustration of the physical model. The vectors represent the different forces that are taken into account to translate the control point $\mathbf{v}_2$. The sphere indicates a collider, and the vector on top shows the current direction of the wind.

- Conservation of length (see Section 5.7)

An illustration of our physical model can be seen in Figure 5.1.

5.1 Force Map

The force map is the general data structure for the physical model and is represented by a two-dimensional texture. Each texel of the force map contains a three-dimensional vector indicating the displacement of $\mathbf{v}_2$ from its original position. The fourth coordinate is called collision force and describes the persistence of an occurred collision. Over time, the value of the collision force decreases until it reaches zero again.

In order to map a texel to a blade of grass, each patch-info object has a squared area assigned for its blades of grass. For this reason, the maximum amount of blades on a single patch is preferred to be near a square of a natural number (see Section 4.3). The position of a square in the force map is defined by a two-dimensional offset vector **o**. This offset vector of a patch-info object depends on its index and on the total amount of patch-info objects. The calculation of the offset vector is shown in Equation 5.1, where *id* is the index of the patch-info object and *p* the total number of patch-info objects.

$$\mathbf{o} = \begin{pmatrix} id \mod \left\lceil \frac{p}{\lfloor \sqrt{p} \rfloor} \right\rceil \\ \frac{id}{\left\lceil \frac{p}{\lfloor \sqrt{p} \rfloor} \right\rceil} \end{pmatrix} \tag{5.1}$$

When a blade of grass wants to read its assigned value from the force map, the texel coordinates have to be calculated. This can be done by knowing the index of the blade of grass, the offset vector of the respective patch-info object and length of the side of a square. Equation 5.2 presents the calculation of the texture coordinates **t**, where i indicates the blade's index and s represents the size of the square.

$$\begin{aligned} \mathbf{t} &= \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \\ t_1 &= s\ o_1 + (i \mod s) \\ t_2 &= s\ o_2 + \frac{i}{s} \end{aligned} \tag{5.2}$$

An example of a force map can be seen in Figure 5.2.

5.2 Stiffness

The stiffness force is always directed towards the idle position of a blade of grass, which makes the blade stand upwards if it is not influenced by any other force. The idle position is defined by both control points being translated from the ground position by the up-vector times the height of the blade.

The strength of the stiffness force is affected by the bending factor of the blade of grass and the current collision force. However, both influences should only decrease the strength and not nullify it. The calculation of the stiffness force **s** is shown in Equation 5.3, where $\mathbf{I}_{\mathbf{v}_2}$ is the idle position of $\mathbf{v}_2$, b the blade's bending factor and c the current collision force.

$$\mathbf{s} = (\mathbf{I}_{\mathbf{v}_2} - \mathbf{v}_2)\left(1 - \frac{b}{4}\right) \max(1 - c, 0.1)\, \Delta t \tag{5.3}$$

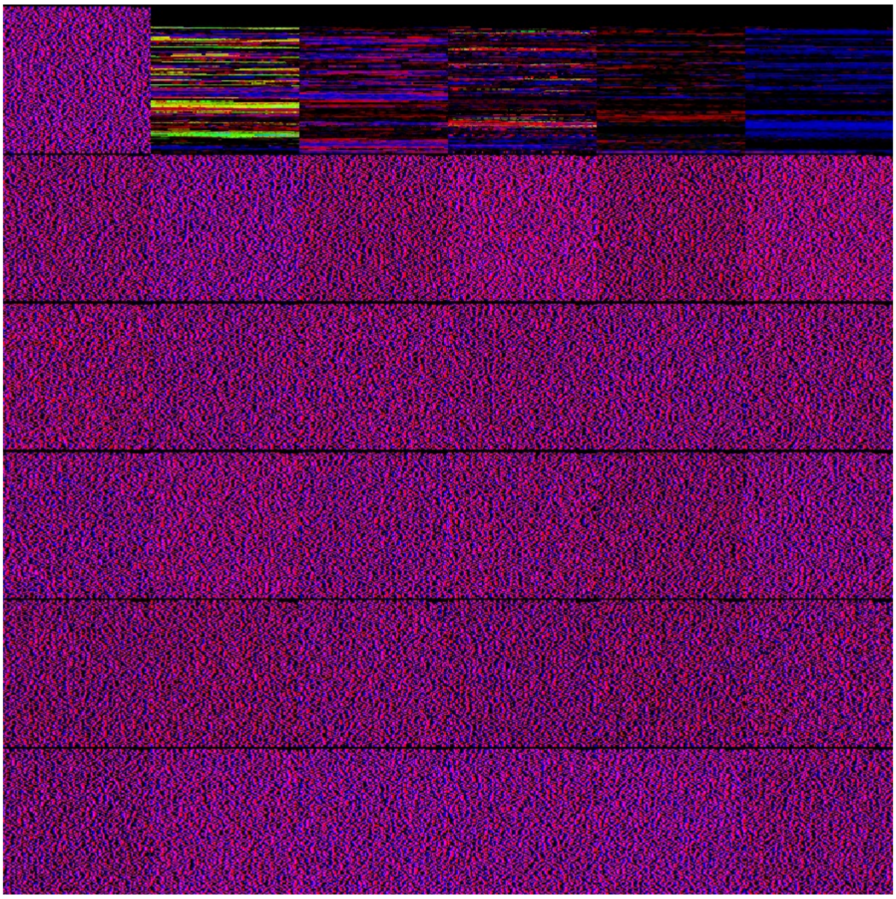

Figure 5.2: This figure shows an example of a force map. Black pixels either refer to only negative displacements or to unused pixels. Thus, the amount of black pixels can be an indicator on the quality of the distribution of the patches. Most of the tiles refer to a meadow aligned along the xz-plane. Therefore, there will be no green displacement of a blade of grass, since this would mean that a blade is torn off the ground by a force. The upper five tiles refer to grass patches on a bunny model. In these tiles all different colors are represented. This can be explained by the blades of grass having a higher variety in their up-vectors, which can lead to displacements in each direction.

5.3 Gravity

Gravity is a constant force that is applied to $\mathbf{v}_2$. The influence of gravity is based on the height of the blade of grass. Small blades are less influenced by gravity than tall blades. The direction and intensity of the gravity can be computed in two different ways.

On the one hand, the direction and intensity can be the same for the whole scene, like in the real world in local areas. On the other hand, the gravity can be pointed towards a specific point in space. This phenomenon also occurs in the real world, seen from a global point of view. In addition, the calculation of the gravity vector can handle a mixture of both variants, which we define as *environmental gravity*, $\mathbf{g}_E$. This interpolation is shown in Equation 5.4, where $\mathbf{G}$ respectively $\mathbf{L}$ represents the global and local gravity vectors with the intensity as the fourth dimension. In addition, t indicates the interpolation parameter between local and global gravity.

$$\mathbf{g}_E = \left(\frac{\mathbf{G}_{xyz}}{\|\mathbf{G}_{xyz}\|} \mathbf{G}_w \, (1 - t) + \frac{\mathbf{L}_{xyz}}{\|\mathbf{L}_{xyz}\|} \mathbf{L}_w t \right) \tag{5.4}$$

If only the environmental-gravity force is applied to a blade of grass, there can be a special case where this gravity force is directed opposite to its up-vector and the blade does not leave its idle state. This is due to the fact that the length correction sets $\mathbf{v}_2$ again to its initial position, if it is just translated along the negative direction of the up-vector. However, since a blade of grass is elastic, the tip of the blade of grass would be pushed downwards in the real world. In order to simulate this effect, we add an additional part to the gravity, which we call the *front gravity*, $\mathbf{g}_F$. This additional force defines a front direction for a blade of grass, which is computed by the cross product between the blade's up-vector and the vector going along the width of the blade. The strength of this force is dependent on the strength of the environmental gravity. Equation 5.5 presents the formula for the front gravity, where $\mathbf{f}$ is the front direction of the blade of grass.

$$\mathbf{g}_F = \frac{1}{4} \|\mathbf{g}_E\| \, \mathbf{f} \tag{5.5}$$

The final vector for the gravity force can be calculated by normalizing the sum of the environmental gravity and of the front gravity. The normalization is done using the height of the blade, the bending factor and the time of the last frame. Equation 5.6 shows the computation of the gravity vector $\mathbf{g}$, where h is the height of the blade, b its bending factor.

$$\mathbf{g} = (\mathbf{g}_E + \mathbf{g}_F) \, h \; b \; \Delta t \tag{5.6}$$

5.4 Wind

In our physical model, the influence of wind depends on four criteria, which are listed below.

- The *direction* of the wind.
- The *strength* of the influence of the wind for a specific position.
- The *alignment* of the blade of grass towards the wind.

- The *bending* factor of the blade of grass.

The strength of the influence of the wind is defined by a three-dimensional wave moving through the space. This wave can either be derived from a direction or a specific point in space. This procedure is similar to the work of Chen et al. [CJ10], who use a numerical model for the dynamics of the continuum to represent grass-grass interactions as wave simulation. However, in our case we evaluate a moving wave function to simulate the strength of the influence of the wind, instead of computing an energy flow. This wave function is an analytic function on a four-dimensional vector and represents the strength of the influence for a given position $\mathbf{p}$. We use the phrase *wind vector* and the greek letter $\vec{\lambda}$ to refer to the four-dimensional vectors, used as input for the analytic functions. In the following sections, we present three different wave functions together with the calculation of the respective direction of the wind.

The alignment criterion indicates how much the blade of grass can be influenced by the wind force. This criterion is developed from two ideas. First, a blade of grass that is standing in its straight position should be influenced more by the wind force than a blade of grass that is pushed to the ground. In addition, if the direction of the force caused by the wind is directed along the width of the blade, the influence of the wind should be less than if the direction of the wind is orthogonal to the blade. Thus, the alignment criterion of a blade of grass depends on the direction of the wind force and the position of $\mathbf{v}_2$ with respect to the blade's up-vector, which leads to Equation 5.7. In this equation, $a\left(\mathbf{d}, h\right)$ is the alignment criterion, $f_d\left(\mathbf{d}\right)$ refers to a factor indicating the alignment of the blade of grass towards the direction of the wind and $\mathbf{d}$ represents the direction of the wind influence. Furthermore, $f_r\left(h\right)$ represents the ratio between the position of $\mathbf{v}_2$ with respect to the up-vector, $\mathbf{up}$ and the height h of the blade.

$$\begin{aligned} f_d\left(\mathbf{d}\right) &= 1 - \left| \frac{\mathbf{d}}{\|\mathbf{d}\|} \cdot \frac{\mathbf{v}_2 - \mathbf{p}}{\|\mathbf{v}_2 - \mathbf{p}\|} \right| \\ f_r\left(h\right) &= \frac{\left|\left(\mathbf{v}_2 - \mathbf{p}\right) \cdot \mathbf{up}\right|}{h} \\ a\left(\mathbf{d}, h\right) &= f_d\left(\mathbf{d}\right) f_r\left(h\right) \end{aligned} \tag{5.7}$$

The four criteria can be combined into an equation to calculate the final vector that indicates the influence of wind on a single blade of grass. This is shown in the following equation, where $\mathbf{d}$ is the direction of the wind influence, $w_{\mathbf{p}}\left(\vec{\lambda}\right)$ refers to the strength of the force at a position $\mathbf{p}$ and b is the bending factor of the blade of grass.

$$\mathbf{w} = \mathbf{d} w_{\mathbf{p}}\left(\vec{\lambda}\right) a\left(\mathbf{d}, h\right) b \, \Delta t \tag{5.8}$$

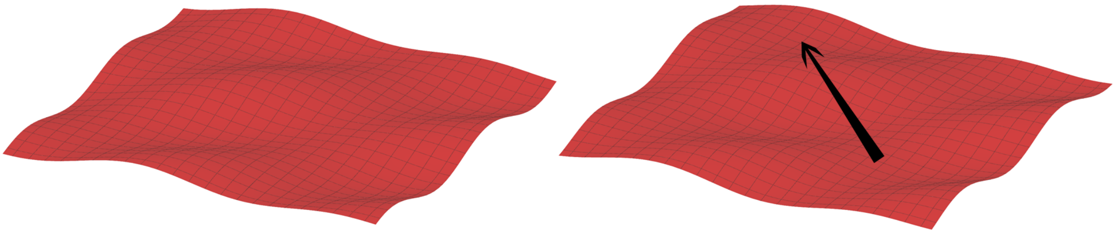

Figure 5.3: This figure illustrates the directional wind influence in two-dimensions. The height of the red surface indicates the influence $w_{\mathbf{p}}\left(\vec{\lambda}\right)$ for a specific position. If the surface is high, the wind has strong influence on the respective position. The right image shows the position-related influence with a higher value of $\vec{\lambda}_w$. The black arrow represents the movement of the wind wave for a better understanding.

5.4.1 Directional wind

The directional-wind function is designed to simulate a casual wind. Depending on the strength of the wind, this function is able to simulate a light breeze or a strong gale. Regarding the wind vector, which is given as input to the function, the first three dimensions of the wind vector indicate both the direction and the strength of the wind. Thus the direction of the wind **d** can be derived directly from the wind vector, as it is shown in the following equation.

$$\mathbf{d} = \vec{\lambda}_{xyz} \tag{5.9}$$

The fourth dimension represents the current position of the wind wave in three-dimensional space. The function for the directional-wind wave consists of overlapping sine and cosine functions and was designed by experimenting. An advantage of this function is the consistent wind influence regardless of the spatial alignment of the field of grass, without a noticeable repetitive pattern.

The calculation of the wind influence for a specific position, $w_{\mathbf{p}}\left(\vec{\lambda}\right)$, is expressed in Equation 5.10. A visualization of this influence is shown in Figure 5.3.

$$w_{\mathbf{p}}\left(\vec{\lambda}\right) = 1 - \max\left(\frac{\cos\left(\frac{3}{4}\left(\mathbf{p}_x + \mathbf{p}_z\right) + \vec{\lambda}_w\right) + \sin\left(\frac{1}{2}\left(\mathbf{p}_x + \mathbf{p}_y\right) + \vec{\lambda}_w\right) + \sin\left(\frac{1}{4}\left(\mathbf{p}_y + \mathbf{p}_z\right) + \vec{\lambda}_w\right)}{3}, 0\right) \tag{5.10}$$

5.4.2 Area wind

The area-wind function refers to a wind deriving from the situation where the source of the wind is a point in three-dimensional space. This influence can be caused by a landing

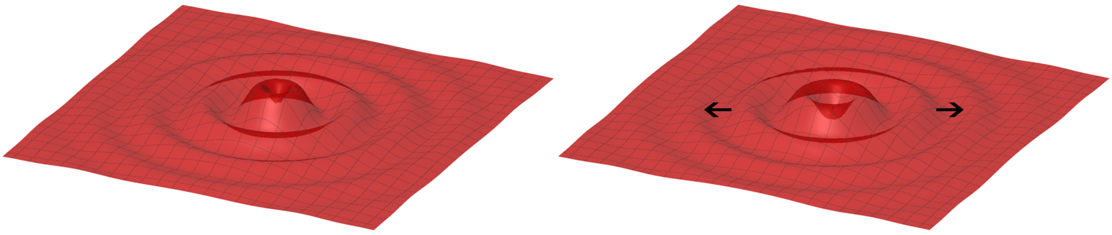

Figure 5.4: This figure illustrates the area wind influence in two-dimensions. The height of the red surface indicates the strength of the influence $w_{\mathbf{p}}\left(\vec{\lambda}\right)$ for a specific position. The position of the wind source is located in the center of the visualization. The right image shows the position-related influence with a higher value of $\vec{\lambda}_w$. The black arrows represent the movement of the wind wave.

helicopter or a hairdryer, for example. Since the wind influence refers to a position in three-dimensional space, the wind vector indicates the source position together with the position of the wind wave. The direction of the wind influence is defined by the position of the blade of grass, $\mathbf{p}$ in relation to the source of the wind, $\vec{\lambda}_{xyz}$. This is shown in the following equation.

$$\mathbf{d} = \mathbf{p} - \vec{\lambda}_{xyz} \tag{5.11}$$

The strength of the influence of the wind depends on two factors. On the one hand, it is evaluated by the analytic function of the wind wave. On the other hand, it decreases as the distance from the blade of grass to the emitter of the wind increases. As in the previous section, the formulas for the the analytic function and the attenuation are designed by experimenting. Equation 5.12 describes the calculation of the attenuation of the strength of the influence, α, by using the length of the wind-direction vector as distance measurement.

$$\alpha = \max\left(1 - \frac{1}{4}\log_2\left(\|\mathbf{d}\| + 1\right), 0\right) \tag{5.12}$$

The final value for the strength of the influence of the wind is calculated by evaluating the analytic function of the wind wave and applying the attenuation factor. Equation 5.13 presents the computation of the strength $\mathbf{w}_{\mathbf{p}}\left(\vec{\lambda}\right)$.

$$w_{\mathbf{p}}\left(\vec{\lambda}\right) = \alpha\left(1 - \max\left(\sin\left(2\left\|\mathbf{d}\left(\vec{\lambda}\right)\right\| - \vec{\lambda}_w\right), 0\right)\right) \tag{5.13}$$

5.4.3 Rotating area wind

The rotating area-wind function is similar to the area-wind function, mentioned in the section above. In contrast to the normal area-wind function, the force is tangential to the wind wave in addition to being directed away from the source. This simulates a rotating wind flow like it is caused by a fan or a tornado. The tangent to the wind wave is calculated in the local plane of the blade of grass, defined by its up-vector. The calculation of the tangent $\mathbf{t}$ is shown in Equation 5.14, where $\mathbf{d}\left(\vec{\lambda}\right)$ is the direction of the wind from the source to the blade of grass and $\mathbf{up}$ refers to the blade's up-vector.

$$\mathbf{t} = \frac{\mathbf{d}\left(\vec{\lambda}\right) \times \mathbf{up}}{\left\|\mathbf{d}\left(\vec{\lambda}\right) \times \mathbf{up}\right\|} \tag{5.14}$$

The rest of the calculation of the wind influence is the same as it is described in the previous section. However, the direction is modified by the tangent, which can be seen in the following equation.

$$\mathbf{d}\left(\vec{\lambda}\right) = \mathbf{d}\left(\vec{\lambda}\right) + \mathbf{t} \tag{5.15}$$

5.5 Ground collision

When strong forces influence a blade of grass, the blade sometimes gets pushed beneath the ground. In order to prevent this phenomenon, the blade of grass has to recognize the ground as collider. Calculating the collision with the ground, represented as a three-dimensional model, multiple times for each blade of grass during the computation of each frame would not be possible in real time. Therefore, we assume that the ground is a plane in the local area of a blade of grass, defined by its up-vector. As a result of this assumption, the check for ground collision can be computed at the cost of a vector dot product.

Since the shape of a blade of grass is defined by a Bézier curve, it can only go beneath the ground if one of the control points is lower than the position of the blade. During the evaluation of the physical model, only $\mathbf{v}_2$ is modified directly. Afterwards, $\mathbf{v}_1$ is set in dependence on the position of $\mathbf{v}_2$ (see Section 5.6). Due to this fact, it is sufficient to ensure that $\mathbf{v}_2$ does not go beneath the ground in order to guarantee a valid state for a blade of grass. This correction procedure is shown in Equation 5.16, where $\mathbf{up}$ is the up-vector of the blade of grass and p indicates the position of the blade.

$$\mathbf{v}_2 = \mathbf{v}_2 - \mathbf{up} * \min\left(\mathbf{up} \cdot \left(\mathbf{v}_2 - \mathbf{p}\right), 0\right) \tag{5.16}$$

5.6 Calculation of v_1

In our physical model, the control point $\mathbf{v}_1$, which defines the curvature, is modified according to the position of $\mathbf{v}_2$. In addition, the position of $\mathbf{v}_1$ is constrained to be on top of the position of the blade of grass, related to its up-vector. According to the fact that $\mathbf{v}_1$ is dependent, only the displacement of $\mathbf{v}_2$ has to be saved for the following frame. Figure 5.5 illustrates the relation between $\mathbf{v}_2$ and $\mathbf{v}_1$.

If $\mathbf{v}_2$ rests in its idle position, which is defined by the position of the blade of grass translated by its up-vector times its height, $\mathbf{v}_1$ and $\mathbf{v}_2$ have equal positions. Otherwise, if $\mathbf{v}_2$ is pushed to the side by the height of the blade, $\mathbf{v}_1$ and the blade of grass share the same position. This behavior is calculated by projecting the vector from the position of the blade to $\mathbf{v}_2$ onto the ground. The length of the projected vector is computed and set in relation to the height of the blade, which indicates the amount of displacement of $\mathbf{v}_1$. This dependency is expressed in Equation 5.17, where $\mathbf{p}$ indicates the position of the blade of grass, $\mathbf{up}$ its up-vector and h its height. In addition, $\mathbf{g_{up}}$ refers to the projection of the vector that points from the position of the blade to $\mathbf{v}_2$ onto the ground plane, defined by the up-vector. The factor 0.05 ensures that $\mathbf{v}_1$ is never perfectly aligned with the position of the blade of grass, in order to always provide at least a slight curvature.

$$\begin{aligned} \mathbf{g_{up}} &= \mathbf{v}_2 - \mathbf{p} - ((\mathbf{v}_2 - \mathbf{p}) \cdot \mathbf{up})\, \mathbf{up} \\ \mathbf{v}_2 &= \mathbf{p} + h\ \mathbf{up} \max\left(1 - \frac{\|\mathbf{g_{up}}\|}{h}, 0.05 \max\left(\frac{\|\mathbf{g_{up}}\|}{h}, 1\right)\right) \end{aligned} \tag{5.17}$$

5.7 Conservation of length

A major drawback of the algorithm of Jahrmann et al. [JW13] is that the length of a blade of grass is not consistent during the animation. On the other hand, that algorithm enabled the blade of grass to be moved freely, which resembles the elastic nature of a blade of grass. Therefore, the goal of the newly proposed physical model is to allow free movement while keeping the length of a blade of grass consistent.

To achieve this goal, the blades of grass are influenced by forces without restriction first, and the length of a blade is corrected afterwards. The problem with this correction is that calculating and correcting the length of a curve precisely for each blade of grass takes too much time. Therefore, the correction algorithm has to be performed as an approximation. According to the work of Gravesen [Gra93], the length of a curve of degree n can be calculated using Equation 5.18. In this equation, L represents the length of the curve, L_0 indicates the distance between the first and the last control point and L_1 is the sum of all distances between a control point and its subsequent one.

$$L = \frac{2L_0 + (n-1)\,L_1}{n+1} \tag{5.18}$$

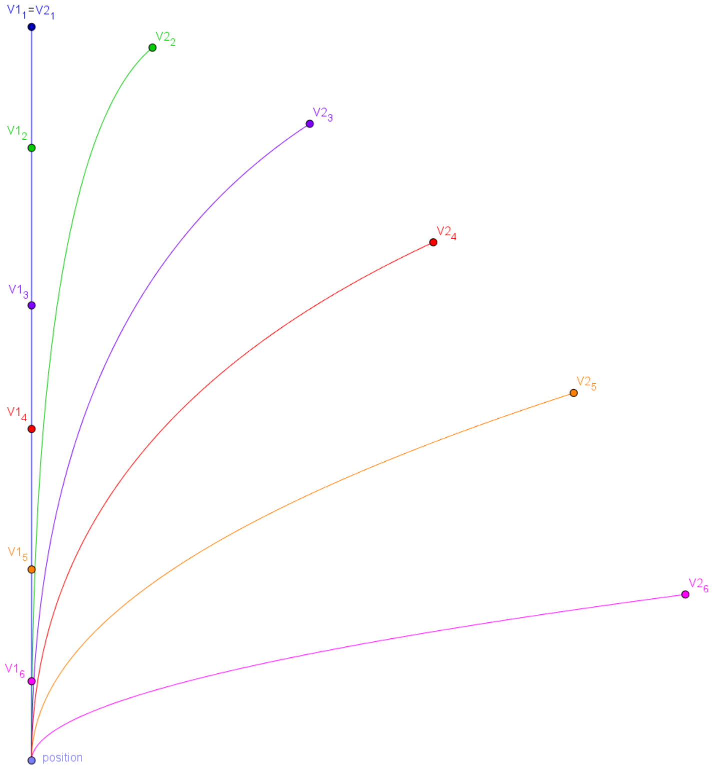

Figure 5.5: This figure illustrates the relation between $\mathbf{v}_1$ and $\mathbf{v}_2$. The different colors symbolize different states of the blade of grass.

Regarding a blade of grass, the Bézier curve has a degree of two, which gives a low approximation error, even for curves with high curvature. However, if more precise results are needed, the curve can be subdivided into two quadratic curves. The subdivision is performed by using the algorithm of De Casteljau [FH00]. In contrast to the usage of a single curve, the two subdivided curves reduce the maximum approximation error by a quarter.

In order to correct the length of the blade of grass, the ratio between the desired length of the blade and the measured curve length is calculated. Finally, the correction of the length is performed by multiplying each segment of the control polygon, defined by the three control points, with the length ratio. This calculation can be seen in Equation 5.19, where h indicates the height of the blade of grass, p the position of the blade and L refers to the measured length of the Bézier curve.

$$\begin{aligned} r &= \frac{h}{L} \\ \mathbf{v}_1 &= \mathbf{p} + r\left(\mathbf{v}_1 - \mathbf{p}\right) \\ \mathbf{v}_2 &= \mathbf{V1} + r\left(\mathbf{v}_2 - \mathbf{v}_1\right) \end{aligned} \tag{5.19}$$

5.8 Sphere collision

In order to simulate natural behavior of a blade of grass, it is important to let the blade react according to its environment. The previous sections have presented natural effects, like wind or gravity, that can be expressed with simple functions. In addition to these effects, we show that our grass-rendering technique is able to evaluate collisions between the blades of grass and the bounding sphere of an object. By using the bounding sphere of an object, only convex objects can generate realistic collision results. However, we also propose a technique for evaluating collisions with complex objects, which is explained in the next section.

In this section, we focus on the interaction between a sphere and a blade of grass. The usage of spheres for collision detection yields benefits over different simple object representations. The benefits that are essential for our algorithm are listed below.

- The test whether a point is inside of a sphere can be evaluated fast by comparing the radius with the distance from the point to the center of the sphere.
- For each point inside of a sphere, the nearest point on the surface of the sphere can be found by simple vector calculations.
- A sphere can be exactly defined by a three-dimensional vector. The first components of the vector represent the position of the center of the sphere and the last component its radius. This minimizes the data that has to be transmitted to the graphics card.

In order to accurately react to collisions with spheres, the collision has to be detected first. This is performed by testing two points of the Bézier curve of a blade of grass whether they are located inside the sphere. However, it is checked first whether the sphere is near enough that a point can be located inside the sphere. This can be done by calculating the distance between the position of the blade of grass and the center of the sphere. This value is compared to the sum of the height of the blade and the radius of the sphere. If the distance is larger, no collision is possible. Equation 5.20 represent this test, where **c** indicates the center of the sphere, r its radius and **p** respectively h refer to the position and the height of the blade of grass.

$$\begin{gathered} d = \left\|\left(\mathbf{c} - \mathbf{p}\right)\right\| \\ d > h + r \rightarrow \text{no possible collision} \end{gathered} \tag{5.20}$$

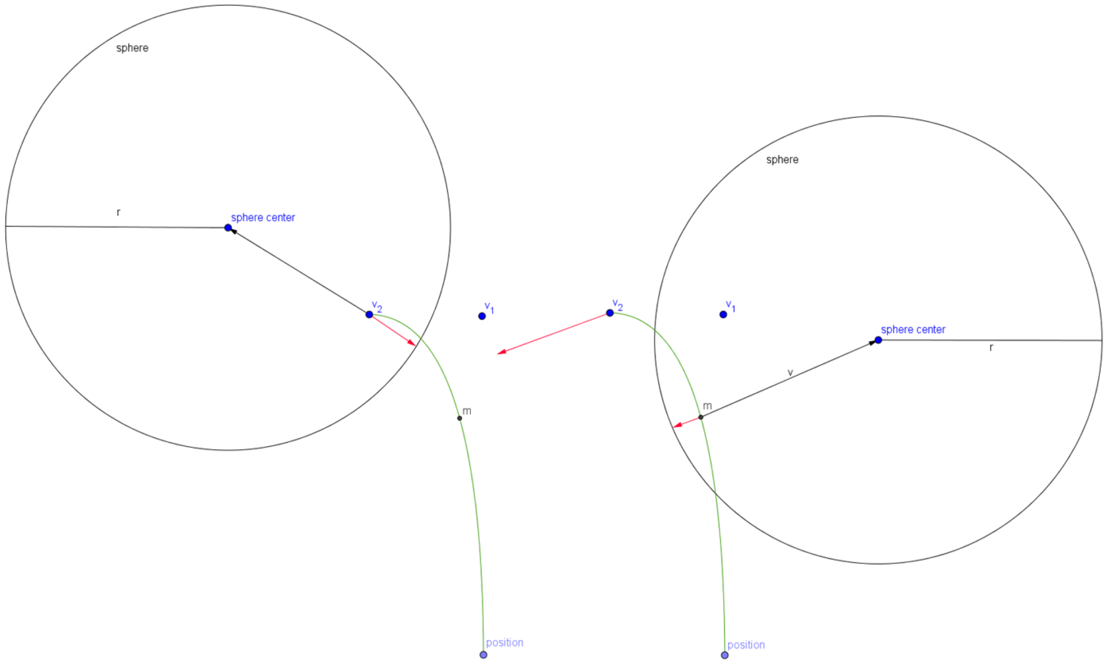

Figure 5.6: This figure illustrates two cases of the collision between a sphere and a blade of grass. In image a) $\mathbf{v}_2$ is inside of the sphere and is moved to the boundary of the sphere. The resulting translation is indicated by the red vector. In image b) the mid point is inside of the sphere and has to be moved to its boundary. Since the physical model modifies $\mathbf{v}_2$ only, $\mathbf{v}_2$ is translated fourfold by the translation of the mid point. These translations are indicated by the two parallel red vectors.

If no collision is possible, no further calculations are needed. This early check is crucial for the performance of our algorithm, since each blade of grass has to be tested with each collision sphere. If a collision is possible, $\mathbf{v}_2$ and the center position of the curve are considered for further calculations. The mid position of the curve can be computed by calculating the spline interpolation with the parameter 0.5. This interpolation is presented in Equation 5.21, where $\mathbf{p}$ refers to the position of the blade of grass.

$$\mathbf{m} = \frac{1}{4}\mathbf{p} + \frac{1}{2}\mathbf{v}_1 + \frac{1}{4}\mathbf{v}_2 \tag{5.21}$$

A collision is detected by our physical model if one of the two points are located inside the sphere. This is an approximation that can lead to undetected collisions if the sphere collider is considerably smaller than the blade of grass. In order to react to this detected collision, the point is moved to the nearest point on the boundary of the sphere. Equation 5.22 presents the translation $\mathbf{t}$ of a point $\mathbf{a}$ inside a sphere with center $\mathbf{c}$ and radius r.

Regarding the collision between a blade of grass and a sphere, this translation is shown in the left visualization of Figure 5.6.

$$\mathbf{t} = \min\left(\|\mathbf{c} - \mathbf{a}\| - r, 0\right) \frac{\mathbf{c} - \mathbf{a}}{\|\mathbf{c} - \mathbf{a}\|} \tag{5.22}$$

However, our physical model is only designed to change the position of $\mathbf{v}_2$. Therefore, if the midpoint is located inside the sphere, its translation to the boundary of the sphere has to be converted to a translation performed on $\mathbf{v}_2$. By analyzing the spline interpolation formula, it can be easily derived that a displacement of the midpoint by a vector $\mathbf{t}$ leads to a translation of $\mathbf{v}_2$ by $4 \cdot \mathbf{t}$. Thus, the translation of $\mathbf{v}_2$ when the midpoint is inside the sphere can be calculated using Equation 5.23, which is also shown in the right visualization in Figure 5.6. In this equation, $\mathbf{m}$ is the midpoint, $\mathbf{c}$ respectively r the center and the radius of the sphere.

$$\mathbf{v}_2 = \mathbf{v}_2 + 4 \min\left(\|\mathbf{c} - \mathbf{m}\| - r, 0\right) \frac{\mathbf{c} - \mathbf{m}}{\|\mathbf{c} - \mathbf{m}\|} \tag{5.23}$$

These collision calculations have to be performed for each blade of grass and each sphere. Each time $\mathbf{v}_2$ is translated due to collision, the squared length of the translation is added to the collision force. This value is saved in the forcemap for the following frames. After all calculations are finished, the three correction steps – ground collision, calculation of $\mathbf{v}_1$ and conservation of length – have to be performed again, to ensure a valid state of the blade of grass.

5.9 Complex object collision

In addition to the collision of simple spheres, the physical model of our approach is also capable of handling collisions with complex models. The difficulty of dealing with complex models lies in the representation of the model used for the calculation. This complexity derives from the fact that the computation of the collision between each polygon of the model with each blade of grass cannot be performed within reasonable time. Therefore, we use spheres as the representation of a complex object. We have chosen spheres because the collision with spheres can be computed fast, as shown in the section above.

Representing a three-dimensional object as spheres is an NP-hard optimization problem, since the problem can be either seen as bin packing problem or set covering problem [AAK$^+$09], which are both NP-hard. There are several variants of this representation problem. Generally speaking, the variants form two groups that are distinguished by the constraint whether the generated spheres are allowed to overlap or not. In this thesis, we choose a sphere representation that does not allow spheres to overlap. This can be solved using a technique called sphere packing [HM09]. However, any other sphere representation should be applicable to our technique as well, which is further discussed in Chapter 10. For finding the spheres, we use the algorithm *ProtoSphere* of Weller et al. [WZ10], which can be implemented on the graphics card. The algorithm consists of an iterative optimization process, where each iteration consists of three steps. However, before the algorithm is performed, a three-dimensional voxel grid has to be precomputed and sent to the graphics card. In this grid, each voxel stores the nearest point on the surface of the model. This data structure represents the model during the optimization loop. In each iteration of the optimization loop, three steps have to be performed, which are listed in the following.

1. One prototype point is generated for each face of the model, which is done on the application side.
2. On the graphics card, the positions of the prototypes are translated n times. For each translation, the prototype finds its nearest surface point, using the voxel grid. When the point is found, the prototype is moved away from the nearest point by its distance to this point. This translation is multiplied by a smoothing factor, which becomes smaller each iteration. After the n translations are calculated, the final distance to the nearest surface point is measured for each prototype.
3. The last step is done once more on the application side. The prototypes are sorted by their distance to the nearest surface point in descending order. Afterwards, a sphere is generated for each prototype if this sphere does not intersect any other spheres. After all prototypes are processed, the voxel grid is updated with the new spheres being treated as additional object boundaries.

In case of our grass-rendering technique, the model needs to be densely packed at the surface and not on the inside. Therefore, the resulting spheres are filtered in a way that only spheres on the boundary of a model remain as the object's representation. This representation can be precomputed for each model once and saved together with the model to avoid long loading times for highly complex models. When our technique is preparing the data for the force update of a patch, the spheres that represent an object are tested for intersection with the bounding box of the patch. All spheres that do not intersect the patch's bounding box are not sent to the graphics card. In addition, spheres that are considerably smaller than the blades of grass of the patch are also filtered out.

Figure 5.7 presents some examples of sphere-packed models.

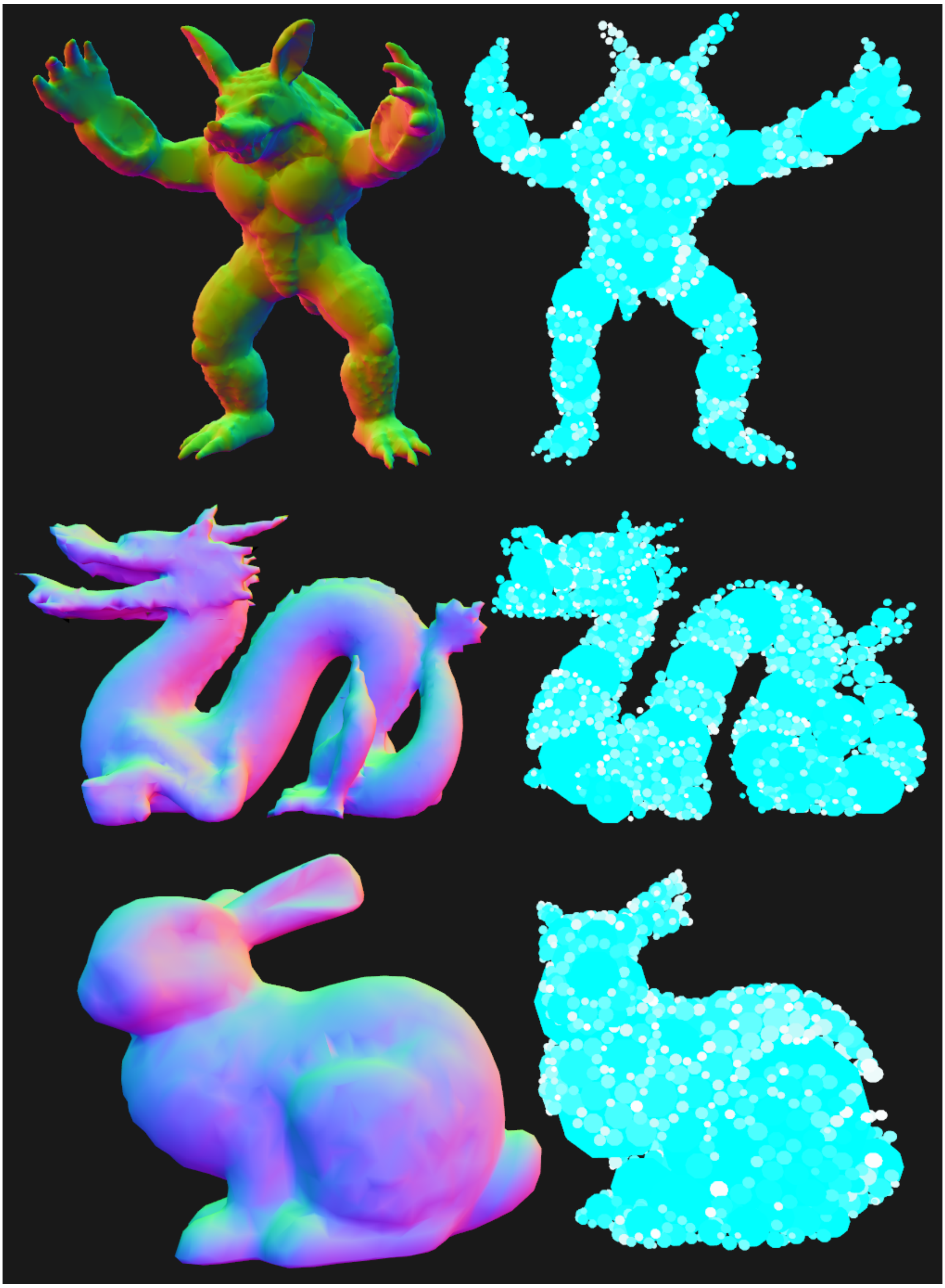

Figure 5.7: This figure shows three examples of sphere-packed objects. The models are taken from the Standford scanning repository [Lab].

CHAPTER 6

Culling Methods

Rendering thousands of blades of grass is a challenging task for the graphics card. Especially, if most of the blades are not visible in the final rendering, this high effort is wasted. Therefore, in this chapter we propose different culling methods varying in accuracy and calculation time. The main goal is to send only those blades of grass to the rendering pipeline, that are really visible in the resulting image. This is achieved by evaluating the visibility of each blade of grass directly on the graphics card using compute shaders. In order to use this visibility information in the redering process, we use a indirect render approach.

The culling on the basis of single blades of grass is an innovative approach, especially in comparison to cited literature, where culling can only be performed on bunches [JSK09, ZLZ09], patches [BPB09, JW13] or other high-level data structures like grids [FLHS15]. The impact on the performance of the culling methods, is presented in Chapter 9.

The following section describes high-level culling, which is executed on application side. In this culling step, the bounding boxes of the fields of grass and the bounding boxes of each patch are tested against the camera's view frustum. Section 6.2 explains low-level culling, which operates directly on the blades of grass. The methods enable culling based on the direction of a blade of grass, on the distance to the camera and on the occlusion by objects. This step is implemented on the graphics card. In addition, the indirect rendering approach and different culling methods are described.

6.1 High-level view-frustum culling

The high-level culling is performed on application side and is responsible for filtering out objects that are completely outside the camera's view frustum. This culling is computed on two different levels of the object hierarchy (see Section 3.1). Each field of grass is represented by an axis-aligned bounding box, which is computed by merging the bounding boxes of the corresponding patches. First, this merged bounding box is tested against the camera's view frustum. If the bounding box is not visible, the whole field is culled. Otherwise, the intersection between the view frustum and the bounding boxes of each corresponding patch is calculated. Each patch, whose bounding box is completely outside the view frustum, is culled.

This view-frustum culling is performed for both update and rendering stages. The update stage consists of the force update and of the visibility update of each blade of grass, which is described in the following sections. The force update is an exception to high-level view-frustum culling. Thus, if a patch's bounding box, respectively the bounding box of a field of grass, is near the view frustum, the force update is computed even if it is not visible. Otherwise, disturbing artifacts can occur if the viewer turns around while the wind is shifting. In this case the patch structure becomes clearly visible, because some patches does not receive the shift of the wind force early enough due to the culling. This unwanted effect is shown in Figure 6.1. More details on the force update are presented in Chapter 5.

6.2 Low-level culling

Low-level culling is performed directly on the graphics card using compute shaders. The main goal is to remove all blades of grass from the rendering pipeline that are not visible in the current frame. In addition, the low-level culling filters out blades, which can cause rendering artifacts. For example, if a blade of grass is seen from the side, some disjointed pixel's can be drawn from this blade of grass, because our representation of a blade has no depth. Therefore, the rendering process has to be flexible enough to possibly render a different amount of blades of grass in each frame. This is achieved throughout the usage of an indirect rendering approach. A detailed explanation of indirect rendering and its application to this grass rendering algorithm is given in the following section.

In Section 6.2.2, we propose different culling techniques. These techniques are responsible for filtering out blades that are unwanted during the rendering process.

6.2.1 Indirect rendering

The common way of rendering objects on the graphics card is performed by direct rendering. In this process, the parameters used for rendering each object are provided by the application and sent to the graphics card. Thus, the application is responsible for determining the amount of objects that are drawn. In case of our grass-rendering technique, only the graphics card has the information about the amount of visible blades of grass. If we were to implement a direct-rendering approach, the application would have to read the visibility results from the graphics card in order to provide the correct parameters for the rendering.

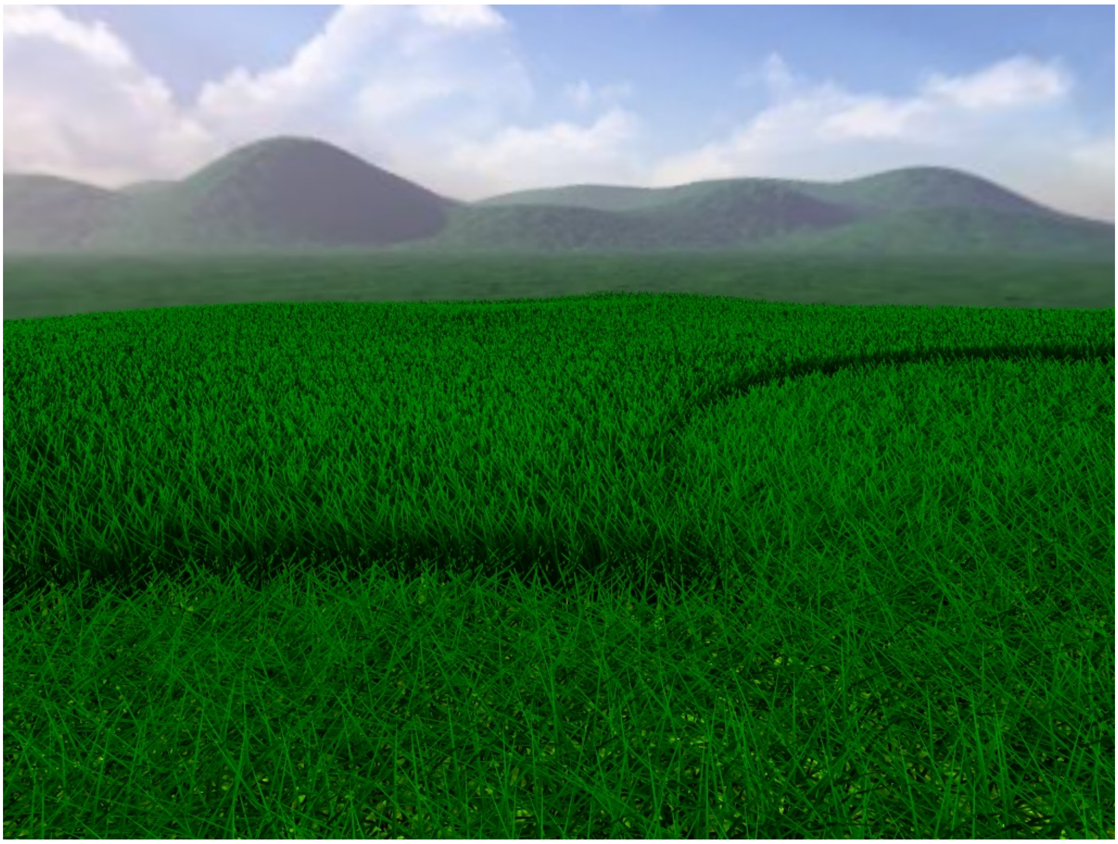

Figure 6.1: This figure shows the artifact that occurs if the force update is not calculated outside of the view frustum. The rendered image is taken after a quick turn of the camera and the scale-formed patch borders can be seen clearly.

Indirect rendering solves this problem. An indirect render call is performed on the application side, like a normal render call. However, the difference is that the parameters of the call are not provided by the application. On the contrary, the parameters are read from a buffer in the memory of the graphics card. Thus, before the rendering starts, a compute shader can perform the culling methods and update the buffer, without having to synchronize with the application. This enables high performance of dynamic rendering.

Indirect rendering is built into all modern graphics application programming interfaces, e.g. OpenGL, DirectX or Vulkan. In this section we focus on the OpenGL implementation of indirect rendering. Nevertheless, the procedure should be similar using other graphics APIs. In the current OpenGL version (4.5), there exist different indirect rendering calls. For our approach, we use an indexed indirect rendering call together with OpenGL's atomic counters. In order to use these techniques, we have to introduce three additional buffers:

- *Atomic counter buffer*: The atomic counter buffer contains a single unsigned integer number. It is responsible for counting the number of visible blades of grass. Detailed information on atomic counters are presented in Section 8.3.2.
- *Index buffer*: The index buffer contains the indices of each blade of grass visible in the current frame. Initially, all indices are listed in this buffer. Afterwards, the buffer is modified during the visibility culling of each frame. When a blade of grass passes all culling methods, it increases the atomic counter value and writes its index to the index buffer.
- *Indirect buffer*: The indirect buffer contains five unsigned integer values. For our approach, only the first value is updated. This value represents the amount of blades of grass, which are drawn in the rendering step.

In order to allow multiple usage of patches for different world positions, each patch being referenced by more than one patch-info object has to perform the rendering directly after the visibility update. Otherwise, the index and indirect buffers can be corrupted with data from another reference of the patch. A sketch of our indirect render approach is shown in Algorithm 6.1.

Algorithm 6.1: Indirect rendering

```
Data: $c$ atomic counter, $idx$ index buffer, $ind$ indirect buffer
for each patch $p_i$ do
    if $p_i$ is visible then
        reset $c$;
        perform visibility tests;
        for each blade $b_j$ do
            if $b_j$ is visible then
                $v \leftarrow increase(c)$;
                set index of $b_j$ to $idx_v$;
            end
        end
        transmit value of $c$ to $ind_{\text{count}}$;
        set OpenGL indirect indexed draw call using GL_PATCHES as type;
    end
end
```

6.2.2 Culling methods

In the following sections, we present different culling strategies to perform culling on each single blade of grass. Besides an increase of performance, these strategies additionally have the goal to enhance the visual appearance of a field of grass by removing blades of grass, which can cause rendering artifacts. All methods are implemented directly on the graphics card using compute shaders. The compute shader dispatches are executed after the force update (see Chapter 5) to have the most recent positions and directions of the blade as input.

During execution, each blade of grass is checked against each enabled culling method. If one of the visibility checks fails, the computation for this blade of grass is finished. Only those blades which pass all checks push their indices into the index buffer and increase the visible blade count.

Orientation culling

The orientation culling method culls blades of grass based on their orientation to the camera. This is important due to the pseudo three-dimensionality of a blade of grass, as it has no thickness. When a blade is rendered from the side, it can cause unwanted aliasing artifacts since its projected pixel width is smaller than the pixel size. Therefore, the direction vector of the camera is compared to the vector going along the width of the blade. This comparison is achieved by calculating the absolut value of the cosine of the angle of the direction vectors, which we name orientation ratio. This can be seen in Equation 6.1, where v is the orientation ratio, $\mathbf{dir_c}$ indicates the direction from the camera to the blade of grass, and $\mathbf{dir_b}$ represents the direction of the blade of grass.

$$v = |\mathbf{dir_c} \cdot \mathbf{dir_b}| \tag{6.1}$$

If the orientation ratio v exceeds a value of 0.9 the blade is culled.

View-frustum culling

During view-frustum culling, each blade of grass is checked whether it is inside the projected frame. Since it is impossible to take each point on the curve of a blade into account, three points are chosen. They are the position of the blade, $\mathbf{v}_2$, and the middle curve point given by the spline interpolation with parameter 0.5. Each of those points is projected to normalized device coordinates by multiplying it with the view-projection matrix. Those normalized device coordinates can be checked whether they are inside the view frustum, by comparing each dimension with the homogenous coordinate, which is represented by the fourth dimension. In addition, it is calculated whether the coordinates are between the camera's near and far plane, using the homogenous coordinate. By calculating only three points, it can occur that segments of the actual blade are visible even if all three points are outside the view frustum. Therefore, we add some tolerance to the variables used in the following calculations. The visibility computation v for a single point $\mathbf{p}$ is shown in Equation 6.2, where $\mathbf{VP}$ represents the view-projection-matrix, t is a tolerance value and n respectively f indicates the camera's near and far plane.

$$
\begin{aligned}
\mathbf{p}_{NDC} &= \mathbf{VPp} \\
\mathbf{p}_{NDC_w} &= \mathbf{p}_{NDC_w} + t \\
n_t &= n - 2t \\
f_t &= f + 2t \\
v &= \mathbf{p}_{NDC_x} > -\mathbf{p}_{NDC_w} \land \mathbf{p}_{NDC_x} < \mathbf{p}_{NDC_w} \\
v &= v \land \mathbf{p}_{NDC_y} > -\mathbf{p}_{NDC_w} \land \mathbf{p}_{NDC_y} < \mathbf{p}_{NDC_w} \\
v &= v \land \mathbf{p}_{NDC_w} > n_t \land p_{NDC_w} < f_t
\end{aligned}
\tag{6.2}
$$

If the outcome of v for each of the three points represents the boolean value $false$, the blade of grass is culled.

Distance culling

When looking across a field of grass, the field becomes more dense near the horizon due to the perspective. During the rendering process, this increase of density introduces two unwanted effects, which are explained in the following.

- The precision of the graphics card to present depth values is higher for small values. Therefore, blades of grass that are far away, can have equal depth values due to the limited precision. This can cause an indeterminism in the rendering pipeline, called z-fighting. The reason is that the graphics card cannot determine which blade is in front of the other, leading to flickering artifacts between different frames of rendering.
- Blades of grass that are far away become smaller due to the perspective. Therefore, at high distances, the blade is smaller than a pixel, which causes aliasing artifacts with neighboring blades of grass.

However, when looking on the field of grass from the top, the density does not increase as much at greater distances, such that less artifacts are visible. In order to solve these problems, the distance-culling method is responsible for culling as many blades as needed to avoid rendering artifacts. Therefore, the distance from the viewer to the blade of grass is projected onto the plane defined by the up-vector of the blade of grass. This does not changes the distances, when looking along the field of grass. However, it decreases the distance when looking perpendicular to the field of grass. This projected distance is calculated in two steps. First, the vector pointing from the camera to the ground position is calculated and projected onto the plane defined by the up-vector of the blade of grass. Afterwards, we compute the length of the projected vector, which gives us the projected distance d_{proj}. This process is shown in Equation 6.3, where $\mathbf{v}_{proj}$ is the projected vector and $\mathbf{c}$ indicates the position of the camera. In addition, $\mathbf{p}$ refers to the position of the blade of grass and $\mathbf{up}$ to its up-vector.

$$\begin{aligned} \mathbf{v}_{proj} &= (\mathbf{p} - \mathbf{c}) - \mathbf{up}\left((\mathbf{p} - \mathbf{c}) \cdot \mathbf{up}\right) \\ d_{proj} &= \|\mathbf{v}_{proj}\| \end{aligned} \tag{6.3}$$

After the calculation is finished, the culling is performed by using the projected distance in relation to a given maximum distance d_{max}. If the projected distance is greater than the maximum distance, the blade is culled immediately. Otherwise, the interval $[0, d_{max}]$ is divided into n levels and the blade is classified to a specific distance level. This classification is shown in Equation 6.4, where l is the resulting distance level. A visualization of the classifiction to the different levels can be seen in Figure 6.2.

$$l = \left\lceil n \frac{d_{proj}}{d_{max}} \right\rceil \tag{6.4}$$

The lowest distance level culls no blades of grass. The second level culls one out of n blade. The higher the level the more blades are culled, until the n^{th} distance level culls $n - 1$ out of n blades of grass. The decision whether a blade is culled is based on the index of the blade and its distance level. The following inequality illustrates the decision process, where id indicates the blade's index. If this inequality is fulfilled, the blade is culled.

$$id \bmod n < l \tag{6.5}$$

The distance-culling method assumes that the blades are sorted in a sense that nearby blades have similar indices. Otherwise, this method can cause holes in a dense meadow.

Occlusion culling

Normally, a rendered scene does not only contain fields of grass, but also other geometrical objects. Thus, blades of grass which are totally hidden by other objects should also be culled. In the following, we propose two methods for detecting hidden blades. Both methods use the same points as the view-frustum culling mentioned above. Those points represent the blade's position, $\mathbf{v}_2$ and the mid point of the interpolation. Some examples of the accuracy of the object-culling methods are presented in Figure 6.3.

Inner-sphere culling: The inner-sphere culling algorithm checks whether a blade of grass is hidden by a maximal sphere contained in a three-dimensional object. Therefore, this culling method works best for convex objects. The maximal inner spheres are precalculated at object generation. In Chapter 8, we present two methods for finding a maximal inner sphere.

Using spheres for the calculation enables easy visibility equations, which can be computed fast and with a low memory footprint. In the same way as the collision spheres of the previous chapter, each sphere is represented by a four-dimensional vector, where three

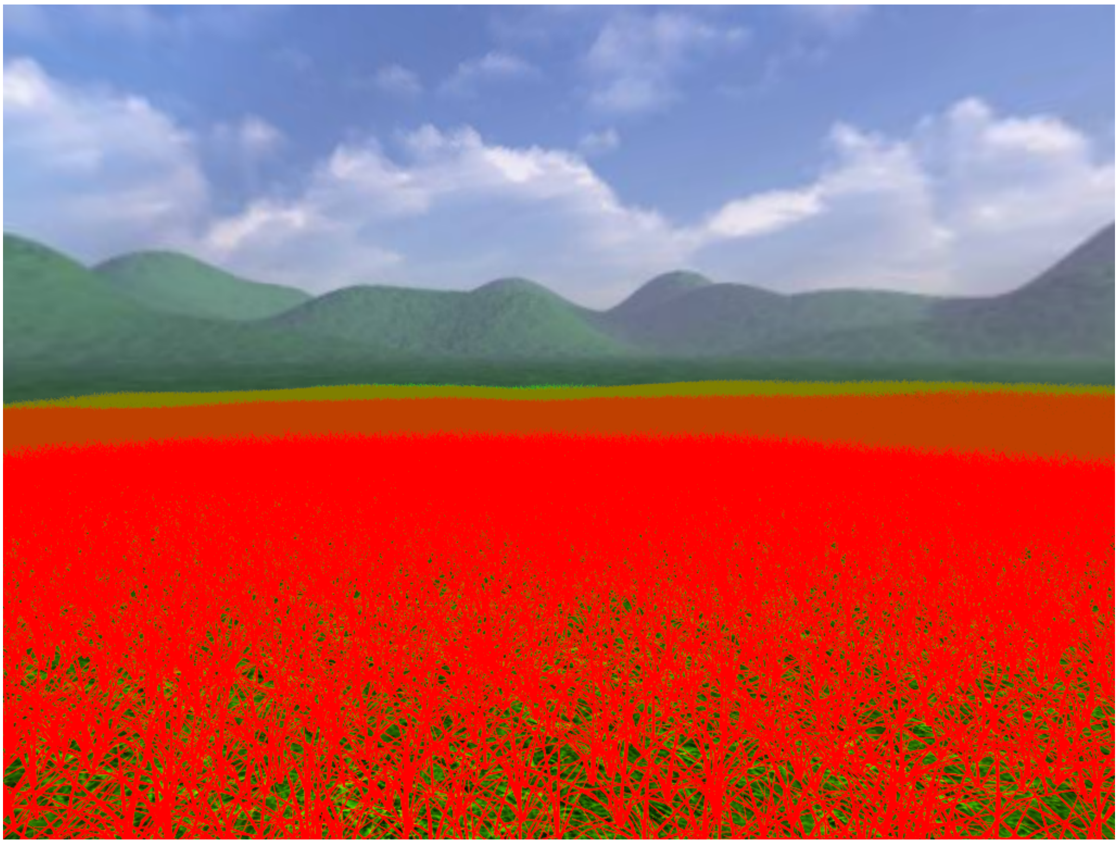

Figure 6.2: This figure illustrates the different levels of the distance culling method. The red color indicates the lowest level whereas the green color represents the highest level.

dimensions indicate the position of the center and the last dimension defines the radius of the sphere. The test whether a blade of grass is hidden behind an inner sphere consists of two steps. This test is performed for each of the three points on the blade of grass. If all of these points are hidden behind the inner sphere, the blade is treated as being not visible.

First, it is checked if the inner sphere is located between the blade of grass and the camera. If this is not the case, there can be no occlusion. This test is computed by projecting the vector from the camera to the center of the inner sphere on the normalized vector from the camera to the blade of grass. If the projected distance of the center of the sphere is smaller than zero or larger than the distance between the camera and the blade of grass, the blade is treated as visible. This is expressed in Equation 6.6, where $\mathbf{p}_c$ is the position of the camera, $\mathbf{p}$ the position of the point on the blade of grass and $\mathbf{s}$ refers to the four-dimensional vector representing the sphere.

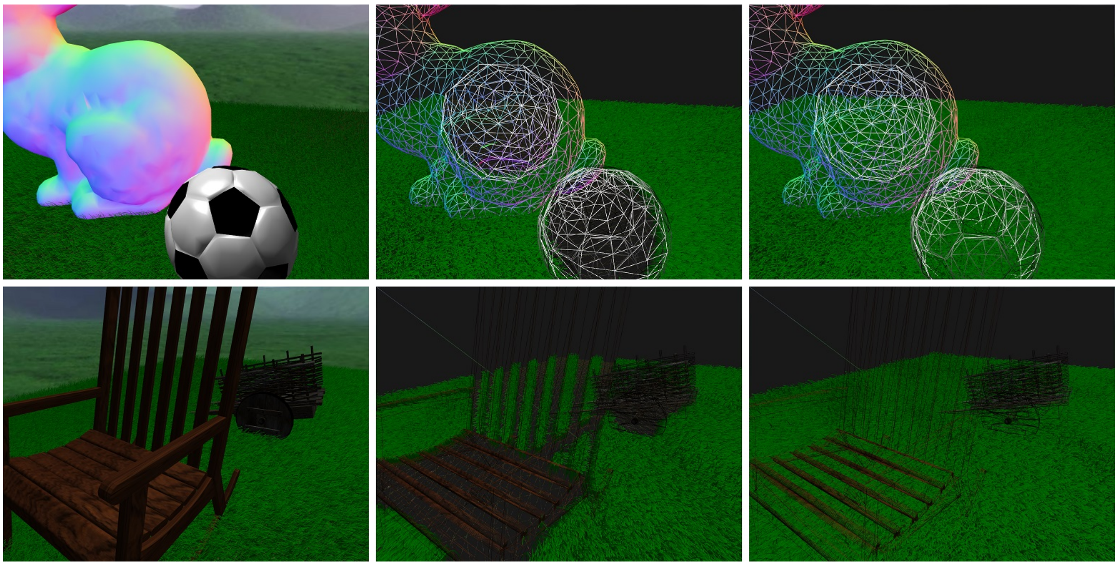

Figure 6.3: This figure shows the accuracy of the occlusion-culling methods. The top row presents examples of inner-sphere culling, whereas the bottom row illustrates depth-buffer culling. The images of the left column show the scenes rendered normally. The middle images show renderings where the wireframe mode is enabled. Finally, the images of the right column present the renderings without culling.

$$0 < \frac{\mathbf{p}_c - \mathbf{p}}{\|\mathbf{p}_c - \mathbf{p}\|} \cdot (\mathbf{p}_c - \mathbf{s}_{xyz}) < \|\mathbf{p}_c - \mathbf{p}\| \tag{6.6}$$

If the shown inequality evaluates to false, the blade is considered as visible and the calculation of this inner-sphere is finished. On the contrary, if the inequality evaluates to true for all three points, the next step of the visibility test is performed. In this step, the shortest distance between the center of the sphere and the line formed by the vector from the camera to the point on the blade of grass is calculated. If this distance is smaller than the radius of the sphere, the point is hidden behind the inner sphere. If this is the case for all three points, the blade is considered as being occluded. This visibility calculation is expressed in the following inequality.

$$\left\| \frac{\mathbf{p}_c - \mathbf{p}}{\|\mathbf{p}_c - \mathbf{p}\|} \times (\mathbf{s}_{xyz} - \mathbf{p}_c) \right\| > \mathbf{s}_w \tag{6.7}$$

Only if this inequality evaluate false for all three points on the blade of grass, the blade is treated as being occluded. The advantage of inner-sphere culling is the calculation that is based on simple equations, which can be calculated fast on the graphics card. However, only convex objects can be good approximated using a sphere.

Depth-texture culling: This culling method uses the depth information of previously rendered objects to find out if a blade of grass is hidden behind an object. The major advantage of this method is that it can handle the occlusion of any non-transparent object regardless of its geometrical properties. However, compared to the inner-sphere culling, the calculation needs more time and an additional depth-texture is needed. In this method, three points of each blade of grass are projected to the screen. Together with the screen size in pixels, the texture-lookup coordinates are calculated and the respective depth values are fetched from the depth texture. Finally, the depth values are compared to the blade's distance to the camera. If the depth value is smaller, the blade of grass is culled. Otherwise, it is not culled. Algorithm 6.2 gives an outline of how depth-texture culling works.

Similar to the problems of shadow mapping [ERC01], unwanted artifacts can appear from aliasing if the sampled depth values refer to surfaces which are not perpendicular to the viewing direction. Therefore, the depth values have to be increased by a small bias.

Algorithm 6.2: Depth-texture culling

```
Data: $D$ depth texture, $\mathbf{c}$ position of the camera, $\mathbf{VP}$ view-projection matrix, $\mathbf{s}$
      two-dimensional vector containing screen size in pixels, $\mathbf{p}_i$ position of blade
      $i$, $\mathbf{v1}_i$ spline control point of blade $i$, $\mathbf{v2}_i$ tip of blade $i$
for each blade of grass $b_i$ do
    $\mathbf{m}_i \leftarrow \frac{1}{4}\mathbf{p}_i + \frac{1}{2}\mathbf{v1}_i + \frac{1}{4}\mathbf{v2}_i$;
    $d_p \leftarrow \|\mathbf{c} - \mathbf{p}_i\|$;
    $d_m \leftarrow \|\mathbf{c} - \mathbf{m}_i\|$;
    $d_{v2} \leftarrow \|\mathbf{c} - \mathbf{v2}_i\|$;
    $\mathbf{p}_{NDC} \leftarrow \mathbf{VP} \cdot \mathbf{p}_i$;
    $\mathbf{m}_{NDC} \leftarrow \mathbf{VP} \cdot \mathbf{m}_i$;
    $\mathbf{v2}_{NDC} \leftarrow \mathbf{VP} \cdot \mathbf{v2}_i$;
    $D_p \leftarrow$ transform depth to distance calculation of
        texelFetch$\left(D, \left(0.5\frac{\mathbf{p}_{NDC_{xy}}}{\mathbf{p}_{NDC_w}} + 0.5\right) * \mathbf{s}\right)$;
    $D_m \leftarrow$ transform depth to distance calculation of
        texelFetch$\left(D, \left(0.5\frac{\mathbf{m}_{NDC_{xy}}}{\mathbf{m}_{NDC_w}} + 0.5\right) * \mathbf{s}\right)$;
    $D_{v2} \leftarrow$ transform depth to distance calculation of
        texelFetch$\left(D, \left(0.5\frac{\mathbf{v2}_{NDC_{xy}}}{\mathbf{v2}_{NDC_w}} + 0.5\right) * \mathbf{s}\right)$;
    if $D_p < d_p \wedge D_m < d_m \wedge D_{v2} < d_{v2}$ then
        cull blade $b_i$;
    end
end
```

CHAPTER 7

Rendering

This chapter deals with the rendering process of a field of grass. The rendering follows a standard forward-rendering approach, which consists of rendering each blade of grass as tessellated object. The naming of the referenced shaders in this chapter refer to the definitions of OpenGL. Other graphics programming interfaces can have different names for the shaders, but the behavior of each stage of the rendering pipeline should be the same.

The procedure is similar to the rendering algorithm by Jahrmann et al. [JW13], with differences in data layout, spatial alignment, shape and coloring. The data layout consists of a single input vertex for each blade of grass with the four-dimensional vectors position, $\mathbf{v}_1$ and $\mathbf{v}_2$ as attributes. The last dimension of the position vector saves the angle of the direction of the blade, the $\mathbf{v}_1$ vector has the blade's height as its fourth dimension and $\mathbf{v}_2$ contains the width of the blade. In addition, a blade of grass is not rectricted to be standing on a plane with the blade's up-vector pointing along the global y-axis. On the contrary, the blade of grass can be aligned in any direction of three-dimensional space. A comparison between the rendering algorithm of Jahrmann et al. and our rendering algorithm can be seen in Figure 7.1.

7.1 Vertex-shader stage

The vertex shader has three major tasks.

- Transformation of the position, $\mathbf{v}_1$ and $\mathbf{v}_2$ from local to world space
- Calculation of the up-vector of the blade of grass
- Calculation of the vector along the width of the blade of grass

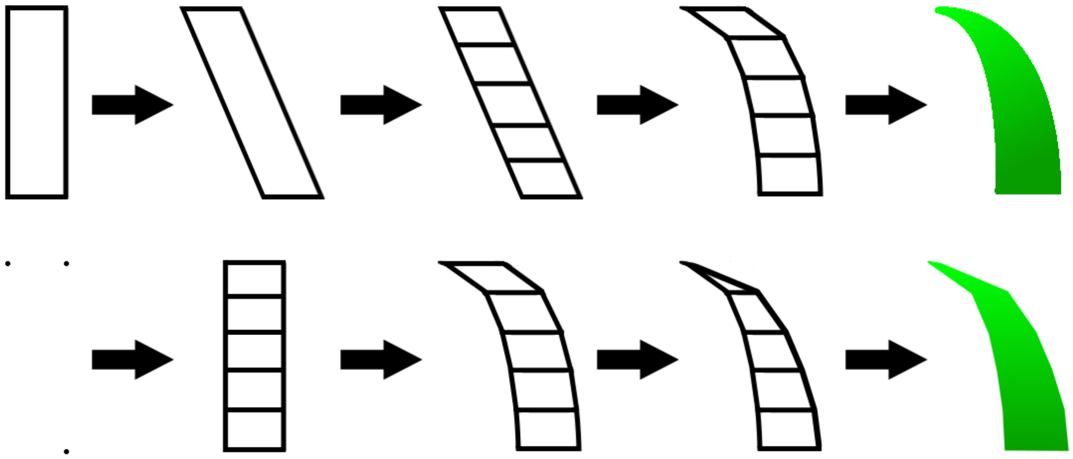

Figure 7.1: This figure shows a comparison between the rendering algorithm proposed by Jahrmann et al. [JW13] and the rendering algorithm presented in this thesis. The top row is extracted from the paper. The first and the second step represent the in- and output of the vertex shader. The third and fourth image shows the input and the result of the tessellation evaluation shader and the last picture outlines a shaded blade of grass. The lower row represents the proposed algorithm. In this row, the first image, showing three points, represent both the in- and the output of the vertex shader. The second to the fourth picture shows the input, the intermediate result before shaping and the output of the tessellation evaluation shader. Finally, the last image outlines a shaded blade of grass of this method.

The transformation from local to world space is done by multiplying the respective three-dimensional vector with a model matrix, using homogeneous coordinates. The up-vector can be easily extracted from the transformed vectors, because $\mathbf{v}_1$ is constrained to be always above the position of the blade of grass, as it is stated in the previous chapter. Therefore, the up-vector is represented by the normalized vector, pointing from the position to $\mathbf{v}_1$. The calculation of the vector along the width of the blade is performed by combining the angle of the direction of the blade with its up-vector. The detailed calculation is described in Equation 7.1, where $\mathbf{v}_w$ indicates the vector along the width of the blade. In addition, d represents the direction value, which is stored in the fourth dimension of the position attribute, $\mathbf{tmp}$ is an auxiliary vector and $\mathbf{up}$ refers to the up-vector of the blade of grass.

$$\begin{aligned}\mathbf{tmp} &= \frac{[\sin(d), \sin(d) + \cos(d), \cos(d)]}{\|[\sin(d), \sin(d) + \cos(d), \cos(d)]\|} \\ \mathbf{v}_w &= \frac{\mathbf{up} \times \mathbf{tmp}}{\|\mathbf{up} \times \mathbf{tmp}\|}\end{aligned} \tag{7.1}$$

7.2 Tessellation shader stage

The tessellation stage is formed by the combination of the tessellation control shader and the tessellation evaluation shader. For the rendering of a blade of grass, a quad is chosen as tessellation primitive. The tessellation control shader calculates the tessellation levels, which represent the number of segments the quad is divided into. The number of segments is essential for the smoothness of the final shape of the blade. Due to perspective, blades of grass that are near the camera are larger than blades that are far away. The goal is to have a uniform smoothness quality of the edges of each blade of grass regardless of the distance to the camera. This is achieved by having a tessellation level for each blade of grass that is proportional to its distance to the camera, which leads a rendering with dynamic level-of-detail. In our application, a four-dimensional vector defines the range of the tessellation levels. It contains the minimum and maximum tessellation level of a blade of grass – l_{min} and l_{max} – together with the minimum and maximum level-of-detail distance, d_{min} and d_{max}. If the distance of a blade of grass to the camera, d, is less or equal to d_{min}, the level has the vaule l_{max}. Otherwise, when d is greater or equal to d_{max}, the level has the value l_{min}. This interpolation is shown in Equation 7.2.

$$level = l_{min} + (l_{max} - l_{min}) \cdot \left(1 - \text{clamp}\left(\frac{d - d_{min}}{d_{max} - d_{min}}, 0, 1\right)\right) \tag{7.2}$$

After the tessellation control shader, the tessellation evaluation shader is executed once for each vertex that is generated by the tessellation process. It is responsible for the final shape of the blade of grass. The bending of the blade is computed by a spline interpolation. This procedure follows the same principles as proposed by Jahrmann et al. [JW13]. In addition, a blade of grass is not allowed to twist around its center axis. This restriction enables that only one spline interpolation needs to be computed instead of two. The interpolation is calculated between the position of the blade of grass, $\mathbf{v}_1$ and $\mathbf{v}_2$, by using De Casteljau's algorithm [FH00]. This algorithm is illustrated in Figure 7.2 and can be seen in Equation 7.3. In this equation, $\mathbf{p}$ stands for the position of the blade, $\mathbf{c}(v)$ is the resulting curve point and v is the curve parameter. A major advantage of De Casteljau's algorithm is the easy calculation of the tangent vector $\vec{t}$ of the blade of grass, since it is given by an intermediate result. The blade's normal vector can be computed by the cross product of the tangent and the bitangent vectors.

$$\begin{aligned}
\mathbf{a} &= \mathbf{p} + v \cdot (\mathbf{v}_1 - \mathbf{p}) \\
\mathbf{b} &= \mathbf{v}_1 + v \cdot (\mathbf{v}_2 - \mathbf{v}_1) \\
\mathbf{c}(v) &= \mathbf{a} + v \cdot (\mathbf{b} - \mathbf{a}) \\
\vec{t} &= \frac{\mathbf{b} - \mathbf{a}}{\|\mathbf{b} - \mathbf{a}\|}
\end{aligned} \tag{7.3}$$

The resulting curve point for each blade of grass is translated by the vector along the width of the blade to get a parallel curve point. Conveniently, this offset vector is equal

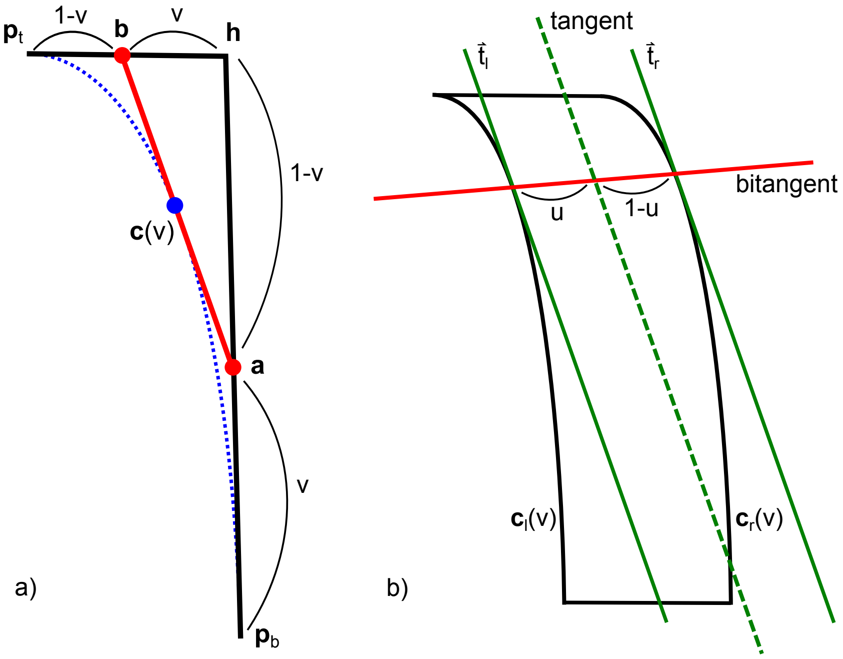

Figure 7.2: This figure is taken from the paper by Jahrmann et al. [JW13]. Image a) shows an illustration of De Casteljau's algorithm for calculating a point on a curve with the parameter v. Image b) presents the calculation of the tangent and bitangent of the blade of grass.

to the bitangent of the blade. By having those two curve points, the final shape of the blade can be calculated. A detailed explanation on the calculation of the shape of a blade of grass is given in Section 7.3.

7.3 Blade geometry

The final shape of the blade of grass is computed using analytic functions. The advantage of analytic functions in comparison to the use of an alpha texture [JW13] is that less fragments have to be calculated if the transparent regions are filtered out in advance by adapting the geometry. Especially, this can improve the rendering performance for blades of grass having a thin tip. In addition, alpha textures can introduce pixelated edges of the blade of grass when being viewed closely. For analytic functions, this effect cannot happen, since the edge of the blade is represented in geometry. The input of our analytic function is defined by six parameters. The output is a three-dimensional vector, which indicates the position of a vertex of the final geometry of a blade of grass. The following list shows the input parameters for the function. For a better understanding, the names of the parameters refer to the names in Figure 7.2, where $\mathbf{c}_1$ and $\mathbf{c}_2$ are two points on the respective curves $c_l(v)$ and $c_r(v)$ and the *normal* is generated by the cross product between the tangent and the bitangent vector.

- $\mathbf{c}_1$: three-dimensional curve-point evaluated with Equation 7.3
- $\mathbf{c}_2$: three-dimensional point being $\mathbf{c}_1$ translated by the bitangent
- u: the blade's horizontal interpolation value
- v: the blade's vertical interpolation value, which has been used in Equation 7.3
- *normal*: the blade's normal vector at the current curve position
- *width*: the width of the blade of grass

By using a subset of these parameters, we propose several blade shapes, which are explained in the following sections. The vertical interpolation value v can have any floating point numbers, whereas the horizontal interpolation value u can only have the distinct values $0, 0.5$ and 1. The values of 0 and 1 indicate the points lying on one of the curves, and a value of 0.5 represents a point on the middle axis of the blade of grass.

7.3.1 Quad, triangle, quadratic and triangle-tip shape

The shape of a blade of grass is formed by four basic shapes. Figure 7.3 presents a sketch of the basic shapes together with the possible values for u and v and illustrates $\mathbf{c}_1$ and $\mathbf{c}_2$ as dotted lines.

Quad: The quad is the most simple shape and is the result of the interpolation between $\mathbf{c}_1$ and $\mathbf{c}_2$ by u.

Triangle: The triangle shape is slightly more complex, since it has to take v into account as well. This gives information about the vertical alignment of the point being calculated. The triangle shape is formed by the interpolation between $\mathbf{c}_1$ and $\mathbf{c}_2$, using the parameter that is calculated by applying the equation $u + 0.5v - uv$.

Quadratic: The quadratic shape is formed like a quad on one side and like a parabola on the other side. This shape can be calculated by the interpolation between $\mathbf{c}_1$ and $\mathbf{c}_2$, using the parameter that is represented by the formula $u - uv^2$.

Triangle-tip: The triangle-tip shape is a combination of the quad and the triangle shape. From the bottom to a given threshold, the blade is shaped like a quad. Further up, it has a triangular shape. This result is reached by interpolating between $\mathbf{c}_1$ and $\mathbf{c}_2$. The parameter of the interpolation is calculated by using the equation $0.5 + (u - 0.5)\left(1 - \frac{max(v-t,0)}{1-t}\right)$, where t is the threshold given as uniform value.

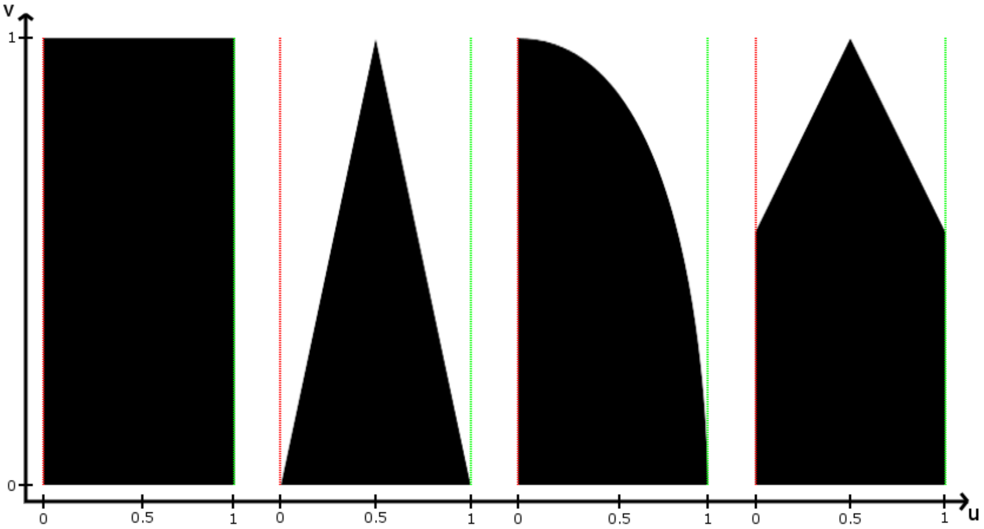

Figure 7.3: This figure shows the four basic shapes that can be applied to a blade of grass. The axes represent the u and v parameters and the dotted red and green lines indicate the positions of $\mathbf{c}_1$ and $\mathbf{c}_2$. The parameter u can have the distinct values 0, 0.5 and 1, whereas v can have any value in the interval $[0, 1]$ depending on the tessellation level.

7.3.2 Three-dimensional shape

In addition to the basic shape, the blade of grass can also become three-dimensional by translating the middle axis along the normal vector of the blade. For shapes having a tip, it is important that the translation has to decrease the nearer the tessellation point is to the top. Otherwise, the blade will have a depth without having a width at the tip. In order to achieve approximately a right angle for the three-dimensional shape, Equation 7.4 can be used. It calculates the translation $\mathbf{t}$, where $\mathbf{n}$ is the blade's normal vector and w the width of the blade. By having a three-dimensional shape, the unfolded width of the blade increases by the factor $\sqrt{2}$.

$$\mathbf{t} = w\ \mathbf{n}\,(0.5 - |u - 0.5|\,(1 - v)) \tag{7.4}$$

7.3.3 Minimal-width shape

When rendering blades of grass at greater distance, especially tipped shapes can be smaller than the size of a pixel. This leads to strong aliasing artefacts. Therefore, we propose the minimal-width shape, which can be added to a basic- or three-dimensional shape. The minimal width is achieved by computing a correction value f that is applied to the parameter of the interpolation between $\mathbf{c}_1$ and $\mathbf{c}_2$. The correction value is calculated by transforming the points on the curves of the blade, $\mathbf{c}_1$ and $\mathbf{c}_2$, to screen coordinates.

For this calculation, the view-projection matrix, **VP** is required. The transformation to screen coordinates is shown in Equation 7.5, where $\mathbf{p}_{\mathbf{c}_1}$ respectively $\mathbf{p}_{\mathbf{c}_2}$ represent the screen coordinates of $\mathbf{c}_1$ and $\mathbf{c}_2$ in pixels.

$$\begin{aligned}\mathbf{p}_{\mathbf{c}_1} &= \frac{\mathbf{VP}\ \mathbf{c}_1}{(\mathbf{VP}\ \mathbf{c}_1)_w} \\ \mathbf{p}_{\mathbf{c}_2} &= \frac{\mathbf{VP}\ \mathbf{c}_2}{(\mathbf{VP}\ \mathbf{c}_2)_w}\end{aligned} \tag{7.5}$$

After the transformation is computed, the width of the blade is measured in pixels. For this, the screen size in pixels, $\begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$, has to be taken into account. Equation 7.6 presents the measurement of the size of the width of the blade of grass between the projected curve points in pixels, which is named $\mathbf{s}_w$.

$$\mathbf{s}_w = \begin{pmatrix} s_1 \left| \mathbf{p}_{\mathbf{c}_{1x}} - \mathbf{p}_{\mathbf{c}_{2x}} \right| \\ s_2 \left| \mathbf{p}_{\mathbf{c}_{1y}} - \mathbf{p}_{\mathbf{c}_{2y}} \right| \end{pmatrix} \tag{7.6}$$

After the width of the blade is measured, the correction value f can be calculated with respect to two constant values, w_{min} and w_{span}. The value of w_{min} indicates the minimum width for a blade of grass in pixels. If a the width of a blade is smaller than or equal to w_{min}, f is equal to one, which enforces that the resulting points have to be equal to $\mathbf{c}_1$ respectively $\mathbf{c}_2$. On the contrary, if f is equal to zero, the interpolation is not influenced at all. For values between zero and one, the interpolation is limited. The second user-defined value w_{span} indicates the interval of pixel sizes in which the interpolation is manipulated by f. In our application, we chose a value of one for w_{min} and a value of two for w_{span}. This enforces a correction for all blades of grass having a projected width in pixel size in the interval of $[0, 3]$. Equation 7.7 shows the calculation of the correction value f.

$$f = 1 - min\left(max\left(\frac{\|\mathbf{s}_w\| - w_{min}}{w_{span}}, 0\right), 1\right) \tag{7.7}$$

7.3.4 Dandelion

In order to show that even complex shapes can be expressed by an analytic function, we propose a mathematical model of a dandelion leaf. Similar to the basic shapes, the dandelion shape is formed by an interpolation between $\mathbf{c}_1$ and $\mathbf{c}_2$. Thus, a suitable interpolation parameter τ has to be found. The function for calculating τ is modeled from a combination of absolute values of sine and cosine functions, which gives the dandelion leaf the spiky shape. Therefore, we have to take the current tessellation level into account in order to not lose any spikes through aliasing. Similar to the functions

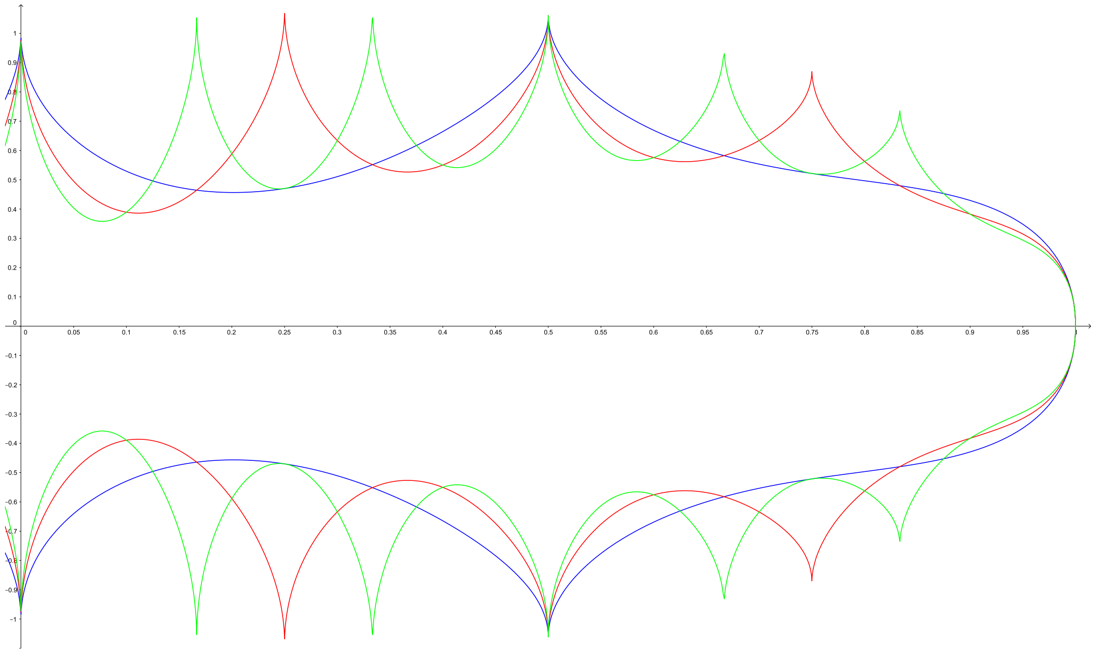

Figure 7.4: This figure shows the graphs of three different levels of the dandelion function. The blue graph corresponds to level 1, the red one to level 2 and the green one to level 3.

of the wind waves, the following formula for the interpolation parameter was developed by experiment to prove that complex shapes can be modeled using analytic functions. This formula is presented in Equation 7.8, where l_π refers to an auxiliary value used for the calculation of τ. In addition, tl indicates the rounded tessellation control shader's tessellation level, given by *gl_TessLevelOuter[0]*, and n indicates a normalization value. The value of n has to be chosen in dependence on the minimum and maximum tessellation level. In our application, we choose the number 8. A plot of three different levels of the dandelion function is presented in Figure 7.4.

$$\begin{aligned} l_\pi &= \pi\ max\left(\frac{tl}{n}, 1\right)(1-v) \\ \tau &= 0.5 + (u-0.5)\left(\sqrt{1-v}\left(1+v-\sqrt{\left|\sin\left(l_\pi\right)\right|\left|\cos\left(l_\pi\right)\right|}\right)\right) \end{aligned} \tag{7.8}$$

7.4 Fragment-shader stage

The fragment shader is responsible for the final shading of the blade of grass. It samples an input texture for the diffuse color and calculates the Blinn-Phong shading model [Bli77] with the ambient-, diffuse- and specular-coefficients, given as uniform values. For rendering young and vivid blades of grass, high values have to be provided for these coefficients. Thus, the final color, especially its green channel, is likely to have values larger than one. In our application, we do not use a high-dynamic-range rendering approach. Therefore, color values that exceed the value of one are clamped by the graphics card. In order not to lose this excessive amount of light energy, we apply a flare effect to the final color. This effect distributes the excess of a channel over one equally to the other color channels. The distribution of the excess can be seen in Equation 7.9, where **c** is the color resulting from the Blinn-Phong shading model. In addition, the dimensions of the three-dimensional vector **e** indicate the excess of a color channel that is distributed to the other channels. An example of the impact of the flare effect on the shading is shown in Figure 7.5.

$$\begin{aligned} \mathbf{e} &= 0.5 \cdot max\left(\mathbf{c} - [1,1,1], [0,0,0]\right) \\ \mathbf{c}_r &= \mathbf{c}_r + \mathbf{e}_g + \mathbf{e}_b \\ \mathbf{c}_g &= \mathbf{c}_g + \mathbf{e}_r + \mathbf{e}_b \\ \mathbf{c}_b &= \mathbf{c}_b + \mathbf{e}_r + \mathbf{e}_g \end{aligned} \tag{7.9}$$

In order to have a higher variation in color, the excessive amount of energy can be calculated by using a threshold instead of the value one. Additionally, factors can be applied to the different color redistributions. These adaptations are especially applicable to artistic rendering scenarios or the simulation of extraterrestrial grass species. Equation 7.10 shows the adapted variations of the equation above. In this modification, **t** indicates the three-dimensional vector for the threshold, and **f** represents the three-dimensional vector for the factors of each color channel.

$$\begin{aligned} \mathbf{e} &= 0.5 \cdot max\left(\mathbf{c} - \mathbf{t}, [0,0,0]\right) \\ \mathbf{c}_r &= \mathbf{c}_r + \mathbf{f}_r\left(\mathbf{e}_g + \mathbf{e}_b\right) \\ \mathbf{c}_g &= \mathbf{c}_g + \mathbf{f}_g\left(\mathbf{e}_r + \mathbf{e}_b\right) \\ \mathbf{c}_b &= \mathbf{c}_b + \mathbf{f}_b\left(\mathbf{e}_r + \mathbf{e}_g\right) \end{aligned} \tag{7.10}$$

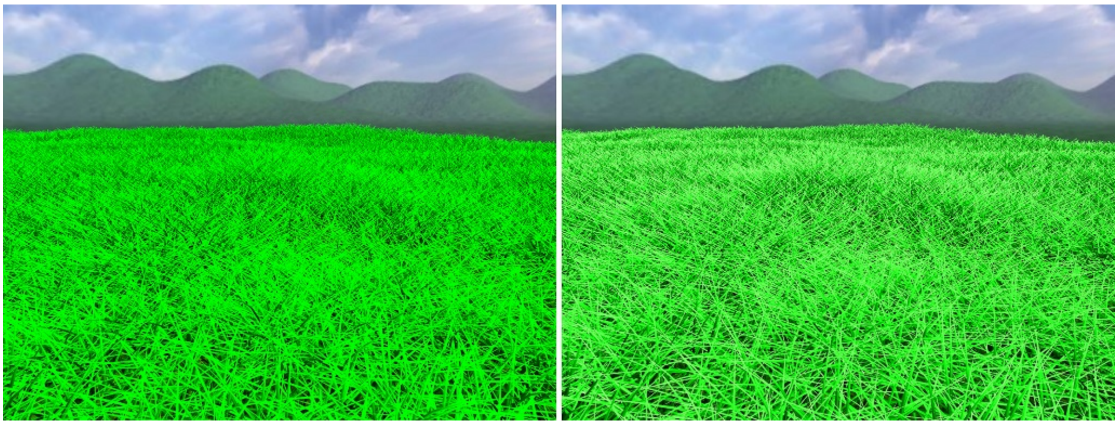

Figure 7.5: This figure shows the different results whether the flare effect is enabled or disabled. The left image shows a frame rendered without flare effect, whereas the right frame is rendered with the flare effect enabled.

CHAPTER 8

Implementation Details

In this chapter, we present some implementation details. The first section introduces the structure of our application, which is relevant for the proposed grass-rendering technique. Section 8.2 presents additional details on the algorithms used in the preprocessing step described in Chapter 4. The following sections describe some OpenGL features used for the implementation, with the goal of either a higher performance or a better usability and extensibility. Finally, the last sections deal with different methods for finding maximum inner spheres, which are used for object culling, described in Chapter 6.

8.1 Application structure

Our grass-rendering technique has the following steps. Each step that is not declared as being processed on the graphics card is performend on application side.

- The three-dimensional model is loaded from a file into the application, and the triangular faces are extracted. These faces are given to the generation process of the field of grass, which is described in Chapter 4. The following section will provide some additional information about the implementation of this step.
- During the rendering loop, the visibility of the patches of a field of grass is tested against the camera's view frustum. In addition, the distance to the view frustum is measured for patches that are outside.
- The physical model is evaluated for each blade of grass on all visible patches and all patches near the view frustum. This step is completely implemented on the graphics card using a compute shader. Detailed information about this step is presented in Chapter 5.

- The visibility is evaluated for each blade of grass that is placed on a visible patch. This step is also performed directly on the graphics card. For the visibility test, we implemented different culling methods, which are presented in Chapter 6. In this step, the amount of visible blades is counted, and the index of each visible blade is written to the index buffer. Additional details about counting on the graphics card and writing to buffers are presented in Section 8.3.2 and Section 8.3.3.

- The rendering of the blades of grass is perfomed using indirect rendering with the parameters provided by the visibility step. An explanation of indirect rendering is given in Chapter 6. Therefore, no dynamic data has to be transmitted to the graphics card for the rendering process. The shape of a blade of grass is defined by an analytic function, which is described in Chapter 7. The branching of the different shapes in the shader is implemented using shader subroutines. A comparison between the use of shader subroutines and common branching is given in Section 8.3.1. For the final shading of a blade of grass, a flare effect is implemented, which is presented in Chapter 7.

8.2 Preprocessing details

In this section, we present additional details for the patch-generation algorithms. Most of these algorithms evaluate distances between three-dimensional points. These distances can be evaluated using different metrics. In our application, we implemented the Euclidean and the Manhattan distance as metrics. Equation 8.1 show the calculation of the distances between two three-dimensional points $\mathbf{a}$ and $\mathbf{b}$, where $d_E(\mathbf{a}, \mathbf{b})$ represents the Euclidian distance and $d_M(\mathbf{a}, \mathbf{b})$ the Manhattan distance.

$$\begin{aligned} d_E(\mathbf{a}, \mathbf{b}) &= \sqrt{\sum_{i=1}^{3} (\mathbf{b}_i - \mathbf{a}_i)^2} = \sqrt{(\mathbf{b} - \mathbf{a}) \cdot (\mathbf{b} - \mathbf{a})} \\ d_M(\mathbf{a}, \mathbf{b}) &= \sum_{i=1}^{3} |\mathbf{b}_i - \mathbf{a}_i| \end{aligned} \tag{8.1}$$

Some example patch divisions of the different algorithms using different metrics are presented in Chapter 9. Apart from distance evaluations, another important task is performed by every patch-generation algorithm, which is the sorting of lists. When dealing with blades of grass, the lists can contain thousands of elements. Therefore, an efficient sorting algorithm has to be used. In our application, we use a multi-threaded version of the quicksort algorithm [Pow91]. The quicksort algorithm is a recursive algorithm that uses the principle of divide-and-conquer. During each recursion, it selects a pivot element and reorders the list such that all elements smaller than the pivot element are on one side and the other elements are on the other side. Then the next step of the recursion is performed both on the list of the smaller elements and on the list of larger elements. After each recursion the pivot element has reached its final position. The multi-threaded version of this algorithm distribute the sorting tasks of the smaller lists to different threads, in order to process lists in parallel.

8.3 OpenGL features

The implementation is based on the graphics application programming interface OpenGL, version 4.5 [SA16]. In the section below, the implementation of shader subroutines is presented together with their advantages. Section 8.3.2 deals with the counting of the visible blades of grass for the indirect rendering approach (shown in Chapter 6). The last subsection presents some information on OpenGL's memory barriers and explains in which parts of our algorithm they are needed.

8.3.1 Shader subroutines

In our application, shader subroutines are used whenever decisions have to be made between different calculation methods. This is done for example for the generation of the shape of the blade of grass in the tessellation evaluation shader (see Section 7.3). Shader subroutines allow branching between different functions at the level of uniform flow control. The benefit is the extensible variety of functions without any additional control structure needed in the shader code. However, each function has to share the same structure, which is a limitation of shader subroutines.

In order to use shader subroutines, four steps have to be accomplished:

- A prototype function has to be stated in the shader code, which defines the in- and output parameters for all subroutine functions.
- A uniform variable with the prototype function as typename has to be declared.
- Subroutine functions have to be implemented with a specific index, defined with the *layout* classifier.
- The application has to provide the index for the desired subroutine, using the command *glUniformSubroutinesuiv*.

In order to compare the difference between the usage of a *switch* control structure and shader subroutines, we implemented some benchmark code with eleven subroutines respectively switch cases. The benchmark works as follows:

1. Both compute shaders have the same eleven tasks.
2. During each iteration, one task is performed. The order of the task follows the list of subroutines respectively switch cases, defined in the shader code.

3. Each task is performed twice and the execute order is reversed for each repetition. This ensures that the execute order of the shader does not manipulate the result.
4. After each iteration, the time needed for the dispatch is measured using OpenGL time queries.
5. At the end of each iteration, the time ratios are written to the console.

The benchmark tests show that the shader using switch-structures is slightly faster when an early case statement evaluates to true. In contrast, the shader using subroutines needs around 15% less time for later tasks. Thus, it can be concluded that shader subroutines have some overhead in comparison to direct branching. However, the overhead is constant and therefore better suited for high branching factors.

8.3.2 Counter objects

On the graphics card, thousands of executions are processed in parallel in different threads. Therefore, the counting process has to ensure that the counting variable is not modified by another thread between the reading of the value and the writing of the result to the variable. Counting is essential for the indirect-rendering approach of our technique, because the amount of visible blades has to be counted and stored in the inderect buffer. An unreliable counting procedure can result in corrupted data to be used during the rendering process. OpenGL is capable of two ways of solving this problem, either by using atomic counters or by using one of the atomic operations on buffer objects.

Atomic counters are special buffers containing only one value. On the one hand, atomic counters are implemented on the graphics card in a highly performant way. On the other hand, the interaction with an atomic counter is limited to three operations, which are listed below.

- *atomicCounter*: This operation leaves the value of the atomic counter unchanged and returns its current value.
- *atomicCounterIncrement*: This operation increments the value of the atomic counter by one and returns the previous value.
- *atomicCounterDecrement*: This operation performs like the increment operation, with the difference that it decrements the value by one.

Resetting an atomic counter to a certain value can be performed in different ways:

- On the graphics card, the value of the atomic counter can be incremented and decremented until it reaches the desired value. Especially, when the counting variable is designed to have high values, this procedure is not recommended. On the other hand, it has the advantage of being implemented completely on the graphics card.

- On application side, the buffer value can be re-uploaded, using the command *glBufferSubData*. This operation needs to pass a single integer value to the graphics card, which can be done relatively fast, regardless of the atomic counter value or the desired value.
- Similar to the procedure mentioned above, the buffer can be cleared by a value using the command *glClearBufferData*. It has the same benefits as the re-uploading of the buffer value. Regarding our grass rendering technique, the performance is the same if the buffer is cleared or the value is re-uploaded.

Modern OpenGL versions also offer an alternative to atomic counters by providing atomic operations. Atomic operations can be performed on any buffer object and are more flexible than atomic counters. In OpenGL version 4.5, there are eight atomic operations, which are listed as follows:

- *atomicAdd*: This operation adds a variable number to the value of the buffer and returns the previous value. This represents an alternative to atomic counters, since it can be used for both incrementing and decrementing.
- *atomicAnd*, *atomicOr* and *atomicXor*: These operations operate on the bitwise representation of a number. They execute the respective Boolean operation and return the previous value.
- *atomicCompSwap*: This operation compares the value of the buffer to a given value. If the comparison is successful, the value of the buffer is updated. In contrast, if the comparison fails, the value is unchanged. In both cases, the previous value is returned.
- *atomicExchange*: This operation sets a specific value to the buffer and returns the previous value. For example, this procedure can be used to reset a counter buffer without the need of transferring data from the application to the graphics card.
- *atomicMax* and *atomicMin*: These operations update the buffer value only if the given value is higher respectively lower than the previous value. In any case, the previous value is returned.

The usage of atomic operations instead of atomic counters can have benefits for our grass-rendering algorithm. Regarding Algorithm 6.1, the atomic counter value needs to be transmitted to the indirect buffer separately after the visibility tests are performed. In contrast, atomic operations can operate directly on the indirect buffer, which reduces the amount of state changes and memory transfer operations. In our application, the performance gain of this modification is not significant. This leads to the assumption that the increment operation of atomic counters outperforms the *atomicAdd* operation.

8.3.3 Memory barrier

During the update process of our technique, there are many steps that manipulate the data of buffers. The evaluation of the physical model updates the buffer that contains the position of the control points of a blade's Bézier curve. The compute shader that is responsible for the visibility calculations needs the updated positions of the control points to correctly evaluate the visibility of each blade of grass. In addition, all visible blades write their index into an index buffer that is used for the indirect rendering.

However, the writing to the memory of buffers or images is not executed immediately on the graphics card. On the contrary, the writing operations are buffered and written at a later time to increase the performance. In our rendering approach, the result of the force update has to be visible in the memory for the visibility update. This is necessary to get accurate results. To allow this, OpenGL provides memory barriers, which can be defined both on application side or inside a compute shader.

A memory barrier can be applied for all types of memory or for a specific type, like atomic counters, shader storage buffer objects or images. Regarding our grass rendering technique, two memory barriers are needed:

- After the force update, a memory barrier for shader storage buffers is needed, since the visibility tests need the information of the shader storage buffers. If subsequent processes would need the updated information of the forcemap, an additional memory barrier for image writings would be necessary.
- After the execution of the compute shader responsible for the visibility, a memory barrier for the atomic counter is needed. This ensures the right value to be transmitted to the indirect buffer. Finally, another memory barrier for shader storage buffers is required. This ensures correct indirect buffer values and index buffer values for the indirect rendering process.

8.4 Maximum inner sphere

In order to have an occlusion sphere for the visibility test (see Section 6.2.2), we propose two methods of finding a maximum inner sphere of an arbitrary three-dimensional model. These methods are shown in the following sections.

8.4.1 Optimization algorithm

At first, we propose a randomized optimization algorithm, which is presented in Algorithm 8.1. The success of finding the maximal inner sphere depends on the number of faces of an object and the number of iterations. During each iteration, a sphere is generated at some position inside of the model. In the following, the maximum radius of the sphere is evaluated. After all iterations are finished, the largest generated sphere is returned as the maximum sphere.

In order to find a valid position for a sphere, a point inside of the three-dimensional object has to be computed. For this purpose, a point-in-polyhedron test is performed, which uses the principle of the non-zero winding rule [Fol96] to determine if a point is inside of a three-dimensional model. This test traces a ray from the point to a random direction and calculates a decision number. For each triangle of the three-dimensional model, the intersection between the ray and the triangle is computed. If the ray intersects a triangle whose normal vector is pointing in the same direction as the ray, the decision number is increased by one. Otherwise, if the ray intersects a triangle whose normal vector is pointing in the opposite direction, the decision number is decreased by one. After all faces are processed, the point is classified as being inside of the polyhedron if the decision number is greater than zero.

The intersection of a ray and a triangle can be computed in various ways. In the following list, we present three different methods, whose calculations refer to Jim Scott [Sco01].

- Barycentric method: The goal of this method is to calculate the barycentric coordinates of the point and to check whether these computed coordinates are valid. The calculation of the barycentric coordinates is shown in Equation 8.2, where u and v are the barycentric coordinates. Moreover, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ indicate the three vertices of the triangle and $\mathbf{P}$ refers to the tested point. If a point is inside the triangle, each barycentric coordinate is positive and the sum of the coordinates does not exceed the value 1. Otherwise, the point must be located outside of the triangle.

- Same-side method: This method performs a check on each edge of the triangle. During each check, the point is tested whether it is on the same side of the edge as the vertex of the triangle, which does not belong to the edge. The point is classified as being inside, if the result of the test of each edge is positive. The test is presented in Equation 8.3. In this equation, $\mathbf{A}$ and $\mathbf{B}$ are the vertices of the tested edge, C indicates the third vertex of the triangle and P refers to the tested point.

- Angle method: The angle method generates vectors from the tested point to each vertex of the triangle and calculates the angle between these vectors. If the sum of the angles is equal to 2π, the point is classified as being inside of the triangle. Otherwise, the point has to be located outside of the triangle. The calculation is shown in Equation 8.4, where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ indicate the three vertices of the triangle and $\mathbf{P}$ refers to the tested point.

$$\begin{aligned}
\mathbf{v}_0 &= \mathbf{C} - \mathbf{A} \\
\mathbf{v}_1 &= \mathbf{B} - \mathbf{A} \\
\mathbf{v}_2 &= \mathbf{P} - \mathbf{A} \\
u &= \frac{(\mathbf{v}_1 \cdot \mathbf{v}_1)(\mathbf{v}_0 \cdot \mathbf{v}_2) - (\mathbf{v}_0 \cdot \mathbf{v}_1)(\mathbf{v}_1 \cdot \mathbf{v}_2)}{(\mathbf{v}_0 \cdot \mathbf{v}_0)(\mathbf{v}_1 \cdot \mathbf{v}_1) - (\mathbf{v}_0 \cdot \mathbf{v}_1)(\mathbf{v}_0 \cdot \mathbf{v}_1)} \\
v &= \frac{(\mathbf{v}_0 \cdot \mathbf{v}_0)(\mathbf{v}_1 \cdot \mathbf{v}_2) - (\mathbf{v}_0 \cdot \mathbf{v}_1)(\mathbf{v}_0 \cdot \mathbf{v}_2)}{(\mathbf{v}_0 \cdot \mathbf{v}_0)(\mathbf{v}_1 \cdot \mathbf{v}_1) - (\mathbf{v}_0 \cdot \mathbf{v}_1)(\mathbf{v}_0 \cdot \mathbf{v}_1)}
\end{aligned} \tag{8.2}$$

$$\begin{aligned}
\mathbf{c}_1 &= (\mathbf{B} - \mathbf{A}) \times (\mathbf{P} - \mathbf{A}) \\
\mathbf{c}_2 &= (\mathbf{B} - \mathbf{A}) \times (\mathbf{C} - \mathbf{A}) \\
\mathbf{c}_1 \cdot \mathbf{c}_2 &\geq 0 \rightarrow \text{inside}
\end{aligned} \tag{8.3}$$

$$\begin{aligned}
\mathbf{v}_1 &= \frac{\mathbf{A} - \mathbf{P}}{\|\mathbf{A} - \mathbf{P}\|} \\
\mathbf{v}_2 &= \frac{\mathbf{B} - \mathbf{P}}{\|\mathbf{B} - \mathbf{P}\|} \\
\mathbf{v}_3 &= \frac{\mathbf{C} - \mathbf{P}}{\|\mathbf{C} - \mathbf{P}\|} \\
\alpha &= \arccos(\mathbf{v}_1 \cdot \mathbf{v}_2) \\
\beta &= \arccos(\mathbf{v}_1 \cdot \mathbf{v}_3) \\
\gamma &= \arccos(\mathbf{v}_2 \cdot \mathbf{v}_3)
\end{aligned} \tag{8.4}$$

8.4.2 Sphere packing

The sphere-packing approach by Weller et al. [WZ10], described in Section 5.9, can also be used to find a maximum inner sphere. The authors state that the first iteration of their algorithm represents the skeleton axis of a model. This is useful, since the largest inner sphere is always centered on the skeleton axis of a model per definition. Therefore, the biggest sphere of the first iteration of the sphere-packing algorithm indicates the maximum inner sphere.

Algorithm 8.1: Maximum inner sphere - optimization algorithm

```
Data: $iterations$ number of iterations, $F$ list of faces, $P$ list of vertex positions
$c_{best} \leftarrow [0, 0, 0]$;
$r_{best} \leftarrow 0$;
while $current\ iteration < iterations$ do
    $p_1 \leftarrow$ select random entry of $P$;
    $p_2 \leftarrow$ select another random entry of $P$;
    $c \leftarrow (p_1 + p_2)/2$;
    if point-in-polyhedron(c) then
        $r \leftarrow \infty$;
        for each face $f_i \in F$ do
            $p \leftarrow$ closest point to $c$ inside triangle $f_i$;
            if $distance\,(p, c) < r$ then
                $r \leftarrow distance\,(p, c)$;
            end
        end
        if $r < r_{best}$ then
            $r_{best} \leftarrow r$;
            $c_{best} \leftarrow c$;
        end
    end
end
```

CHAPTER 9

Results

In this chapter, we show the results of our proposed rendering technique and compare them to related algorithms. The evaluation of our results is based on visual appearance, elapsed time on the graphics card and frames per second.

Our test application is implemented in C++ and uses OpenGL, version 4.5. The results are generated on a machine using an NVIDIA GeForce GTX 780M graphics card and an Intel Core i7-4800 @ 2.7 GHz CPU with 32 GB RAM. The screen resolution for the renderings is 1024 x 768 pixels.

In the first section, we present the results of the different patch-generation algorithms. Section 9.2 introduces our test scenes and evaluates them. Finally, we discuss the results and compare them to related algorithms.

9.1 Patch generation results

In this section, we compare the different patch-generation algorithms. Since this step is processed completely on the CPU, we evaluate the elapsed time by comparing CPU timestamps. The evaluation is based on two fields of grass, which are described in the following sections. In the first scenario, we test the generation of the patches on a flat field of grass. In comparison, the second scenario evaluates the patch-generation process on a three-dimensional model.

The evaluation is done in three ways. First, the division is shown in a figure, and the shape of the resulting patches is discussed. Afterwards, a table lists the evaluation results. The measured values for each patch-generation method are the time required for the generation of the field of grass and the compactness of the patches. The compactness of the resulting patches is represented by the mean square distance MSD from a blade to the center of the corresponding patch. The mean square distance can be calculated using the following equation, where $\mathbf{b}_i$ is the i^{th} blade of grass and $\mathbf{c}_j$ refers to the center of patch P_j.

$$MSD = \frac{1}{n} \sum_{j=0}^{m} \sum_{\mathbf{b}_i \in P_j} (\mathbf{c}_j - \mathbf{b}_i) \cdot (\mathbf{c}_j - \mathbf{b}_i) \tag{9.1}$$

9.1.1 Flat-surface division

The field of grass of the first test scenario is generated on a flat surface and contains 400,000 blades of grass, which are divided into 27 patches. For this test scenario, the blades of grass are distributed by the single-blade seeding. The resulting divisions can be seen in Figure 9.1. The single image in the first row presents the result of the largest-dimension-sorting algorithm. Below, the first column shows the seed-point algorithm using the box-subdivision method for generating the seed points. The second column represents the seed-point algorithm using the factorization method for generating the seed points. The third column illustrates the results of the clustering algorithms.

The rows represent the different variants of the algorithms according to the respective column. The variants of the seed-point algorithms are described in the following.

1. Normal seed-point algorithm using the Euclidean distance metric.
2. Normal seed-point algorithm using the Manhattan distance metric.
3. Advanced seed-point algorithm using the Euclidean distance metric.
4. Advanced seed-point algorithm using the Manhattan distance metric.

The variants of the clustering algorithms are listed below.

1. Nearest-neighbor clustering using the Euclidean distance metric.
2. Nearest-neighbor clustering using the Manhattan distance metric.
3. Final result of the k-means clustering algorithm using the Euclidean distance metric.
4. Final result of the k-means clustering algorithm using the Manhattan distance metric.

When analyzing the images from Figure 9.1, each method generates patches of different shape. Only in four images, all patches are completely connected. As to be expected, the result of the k-means clustering algorithm provides connected clusters regardless of the used distance metric. In addition, the nearest-neighbor clustering using the Euclidean distance metric and, obviously, the largest-dimension sorting algorithm generate connected clusters. Every other method produces at least one disjointed cluster.

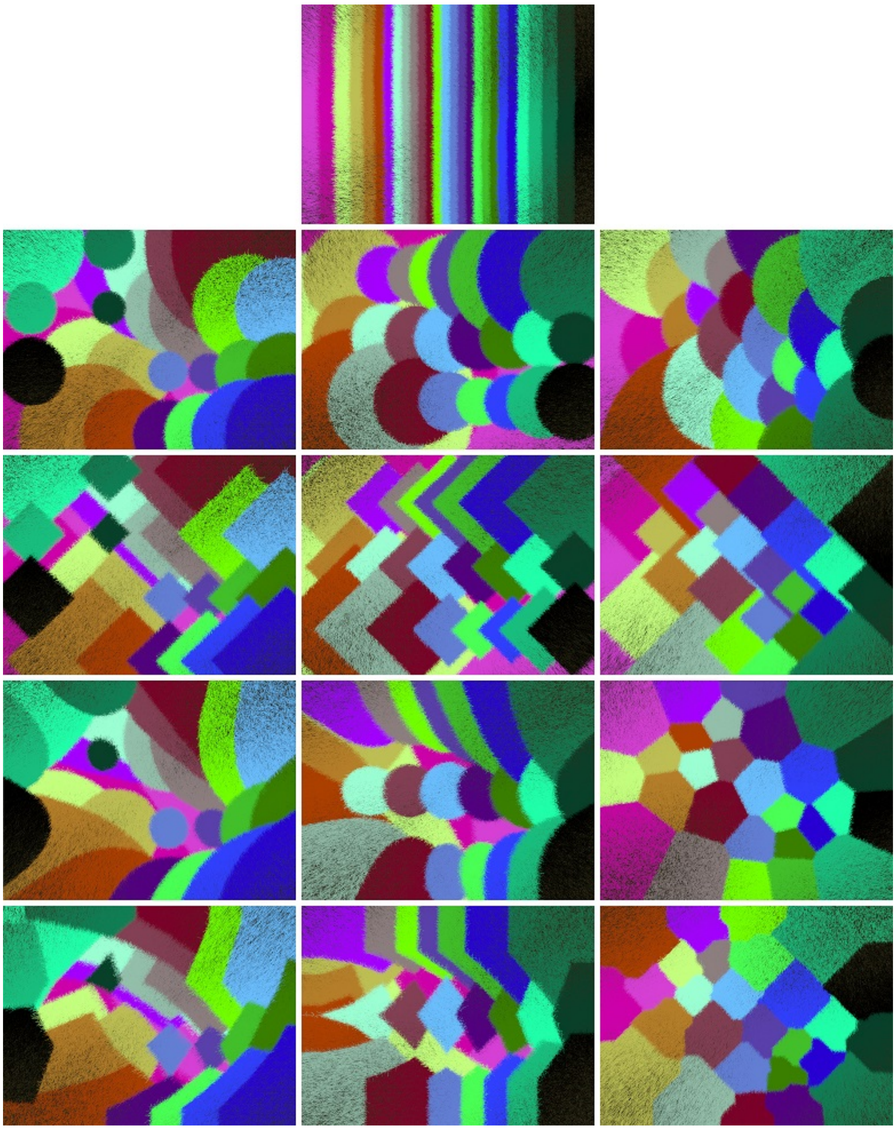

Figure 9.1: This figure shows the resulting patch divisions of the different patch-generation algorithms of the flat-surface division. The topmost image is generated by using the largest-dimension sorting algorithm. Regarding the images below, the columns present the methods as follows: the seed-point algorithm using the box-subdivision method, the seed-point algorithm using the dimension-factorization method and the clustering algorithms. The rows for the first two columns represent the normal seed-point algorithm using the Euclidean and the Manhattan distance metric. Below, the images show the advanced seed-point algorithm using the Euclidean and the Manhattan distance metric. The last column shows the different variants of the clustering algorithms. The first two rows represent the nearest-neighbor clustering algorithm using the Euclidean and the Manhattan distance metric. The last two rows present the results of the k-means clustering algorithm using the Euclidean and the Manhattan distance metric.

Apart from the visual evaluation of the shapes of the generated patches, we compare the running time of the different methods and the compactness of the generated patches. This comparison is presented in Table 9.1. The table shows that the clustering algorithm is clearly the slowest patch-generation method. This outcome was expected since it is the most complex algorithm and generates the most compact clusters. However, if less compact clusters are sufficient, the nearest-neighbor clustering has the best ratio between compactness of the generated patches and time consumption.

Method	Time (s)	MSD
Largest-dimension sorting	1.872	96.587
Normal box-subdivision with Euclidean metric	2.699	30.257
Normal box-subdivision with Manhattan metric	2.437	30.329
Advanced box-subdivision with Euclidean metric	4.024	23.889
Advanced box-subdivision with Manhattan metric	3.057	29.952
Normal dimension-factorization with Euclidean metric	2.713	48.921
Normal dimension-factorization with Manhattan metric	2.556	51.757
Advanced dimension-factorization with Euclidean metric	4.427	23.915
Advanced dimension-factorization with Manhattan metric	3.241	22.562
Nearest-neighbor clustering with Euclidean metric	1.302	17.067
Nearest-neighbor clustering with Manhattan metric	1.347	17.305
K-means clustering algorithm with Euclidean metric	17,171.823	12.759
K-means clustering algorithm with Manhattan metric	19,851.525	13.473

Table 9.1: This table represents the evaluation of the flat-surface division. The running time of the generation of a field of grass using different patch generation algorithms is measured. Likewise, the compactness of the patches is quantified by the minimum square distance of the resulting patches.

9.1.2 Complex-model division

In this scenario, the field of grass is placed on the surface of a three-dimensional object. We have chosen the bunny model from the Stanford scanning repository [Lab]. The field of grass contains 203,945 blades, which are divided into 14 patches. The representation of the results is the same as in the previous section. Figure 9.2 shows the resulting divisions.

In comparison to the distribution of blades of grass on a flat surface, the different patch divisions on a complex three-dimensional model generated by the different methods are much more similar. This is also reflected in the mean square distance (MSD) for the different divisions, which is shown in Table 9.2. The span between the best and the worst MSD value is narrower than in case of distributing on a flat surface. Only one method is clearly outperformed in comparison to the others. The simple version of the seed-point algorithm using the dimension factorization method certainly has the largest mean square distance values. This might be due to the reason that 14 has only two possible factorizations with three factors. However, the advanced seed-point algorithm of

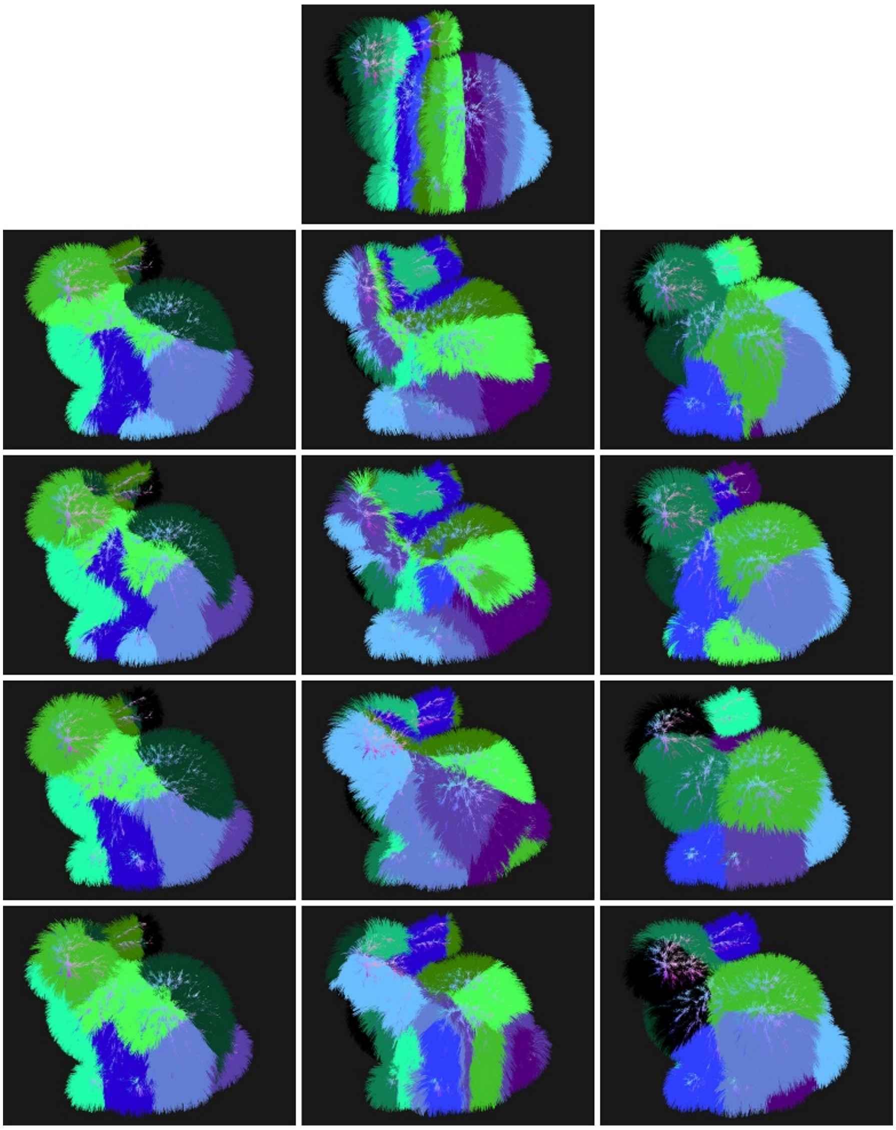

Figure 9.2: This figure shows the resulting patch divisions of the different patch-generation algorithms of the complex-model division. The topmost image is generated by using the largest-dimension sorting algorithm. Regarding the images below, the columns present the methods as follows: the seed-point algorithm using the box-subdivision method, the seed-point algorithm using the dimension-factorization method and the clustering algorithms. The rows for the first two columns represent the normal seed-point algorithm using the Euclidean and the Manhattan distance metric. Below, the images show the advanced seed-point algorithm using the Euclidean and the Manhattan distance metric. The last column shows the different variants of the clustering algorithms. The first two rows represent the nearest-neighbor clustering algorithm using the Euclidean and the Manhattan distance metric. The last two rows present the results of the k-means clustering algorithm using the Euclidean and the Manhattan distance metric.

this variation has a much lower MSD value. This improvement is also clearly visible in the respective images of Figure 9.2. It shows that the patch borders for the advanced seed-point algorithm using the dimension-factorization method are considerably less indistinct than in case of the simple seed-point algorithm.

Regarding the k-means clustering approach, the generated patches are again the most compact patches, regardless of the used distance metric. However, the nearest-neighbor clustering and the seed-point algorithm using the box-subdivision approach generate promising patches in a fraction of the time required for the accurate k-means clustering. Like in the case of a flat field of grass, the nearest-neighbor clustering algorithm provides the best ratio between running time and compactness of the generated patches.

When analyzing the mean square distance values of all approaches, it can be clearly noticed that the Euclidean distance metric is better suited for finding patches on top of a complex three-dimensional object. In comparison, both distance metrics have more or less similar results in the case of a flat field of grass.

Method	Time (s)	MSD
Largest-dimension sorting	0.751	26.298
Normal box-subdivision with Euclidean metric	1.275	9.168
Normal box-subdivision with Manhattan metric	1.167	11.691
Advanced box-subdivision with Euclidean metric	1.513	9.513
Advanced box-subdivision with Manhattan metric	1.235	11.011
Normal dimension-factorization with Euclidean metric	1.331	20.865
Normal dimension-factorization with Manhattan metric	1.317	22.208
Advanced dimension-factorization with Euclidean metric	1.487	12.538
Advanced dimension-factorization with Manhattan metric	1.321	13.060
Nearest-neighbor clustering with Euclidean metric	0.384	8.935
Nearest-neighbor clustering with Manhattan metric	0.394	9.558
K-means clustering algorithm with Euclidean metric	4,045.793	5.775
K-means clustering algorithm with Manhattan metric	3,233.395	6.060

Table 9.2: This table represents the evaluation of the complex-model division. The running time of the generation of a field of grass using different patch generation algorithms is measured. Likewise, the compactness of the patches is quantified by the minimum square distance of the resulting patches.

9.2 Scenes and evaluation

In this section, we show examples of our rendering technique. In order to show the flexibility of our approach, we introduce several different scenes and evaluate the performance of our technique. The evaluation is based on different measurements, which are stated below. The timing is done on the graphics card using query objects. The time required for the update process consists of two parts. First, the sum of the time used for the force update of all patches near the view frustum. Second, the time needed for the visibility update of all patches inside the view frustum. Both parts are also evaluated separately for a better comparison.

- FPS: rendered frames per second
- B_d: number of blades drawn
- B_c: number of blades culled
- T: total time for rendering the whole scene
- T_u: total time used for the update process of all visible patches in miliseconds
- T_f: total time used for the force update of all visible patches in miliseconds
- T_v: total time used for the visibility update of all visible patches in miliseconds
- T_r: total time used for the rendering process of all visible patches in miliseconds
- S: number of collision spheres processed by the force update

The evaluation of the scenes is based on 8 measurements. In order to guarantee a reasonable comparison, all measurements are taken from frames having the exact same input data as it can be seen in the renderings. The options for the different measurements are listed below.

- #1: All implemented features are enabled
- #2: No distance culling
- #3: No low-level view-frustum culling
- #4: No inner-sphere culling
- #5: No depth-buffer culling
- #6: No orientation culling
- #7: No collision detection
- #8: All implemented features are disabled

In the following sections we present the different scenes and their evaluation. The discussion of the results of the scenes is shown in Section 9.3.

9.2.1 Nature scene

The nature scene consists of several objects and resembles an outdoor scenario. The goal of this scene is the integration of grass rendering into a natural scenario. The field of grass is placed on a terrain with smooth hills. It has 397,881 blades of grass, which are divided into 26 patches. We use nearest-neighbor clustering with the Euclidean metric for the patch generation. In addition, the scene contains a bunny model, which is represented as collision spheres. The effect of the physical model is shown by two rolling balls, which leave a trail behind. Additionally, two objects are added, for a better visual representation. Both balls contain inner spheres, and also the largest sphere of the bunny is used as maximum inner sphere.

Figure 9.3 shows some renderings of the scene, and the measurements of the results are presented in Table 9.3. In the table, we highlighte the most interesting measurement results. It can be seen clearly that almost three-fourths of all the blades of grass of visible patches are culled if all culling methods are enabled. Nevertheless, the field of grass has a dense appearance without any bare spaces.

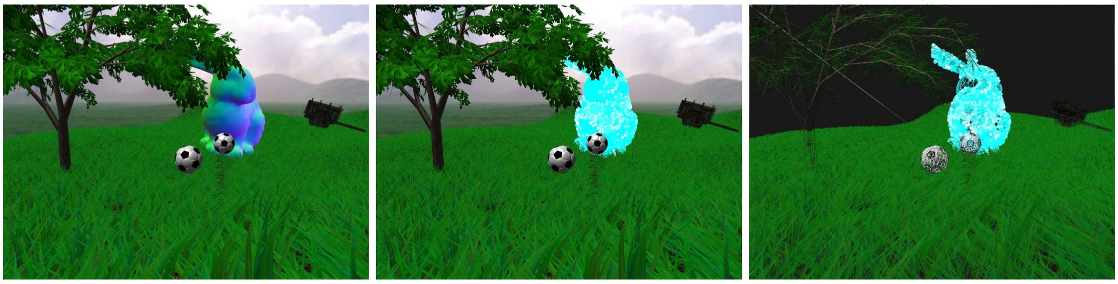

Figure 9.3: This figure shows renderings of the nature scene. The left image shows a normal view of the scene. The middle image presents the collision spheres of the bunny model, which are generated by the sphere packing algorithm. The right image represents a wire-frame view of the scene. It can be seen that the tree is too thin to completely hide blades behind it. However, the culling of the other objects can be noticed easily in the this rendering mode.

An interesting phenomenon is presented in case of disabling the low-level view-frustum culling. The time consumption of the visibility test is significantly higher than with the enabled view-frustum culling. This can be explained by the order of execution of the different culling methods. The view-frustum culling is the first culling method that is executed. Therefore, all blades of grass that would have been culled by view-frustum culling have to pass all other culling tests.

By disabling the inner-sphere culling, the amount of visible blades stays the same. This is no coincidence, since each blade that is hidden behind an inner sphere also has to be hidden behind the object, which is detected by the depth-buffer culling. In contrast to disabling view-frustum culling, the time consumption of the visibility test increases only slightly. This occurs due to the fact that inner-sphere culling is almost at the end of the execution order of the culling methods. Thus, blades of grass that would have failed the visibility test due to inner-sphere culling only have to pass a few additional tests.

The culling method that has the biggest impact on the number of culled blades of grass is the direction culling. More than 15,000 blades of grass are exclusively culled by this method, which represents about ten percent of all blades of grass on visible patches.

Another interesting measurement of the visibility update is that the time consumption is higher if no blades are culled. This can be explained by the fact that the visibility compute shader is also responsible for counting the amount of visible blades and filling the index buffer. Thus, the fewer blades are culled, the more data has to be written to the buffers, which leads to an increase of time consumption.

Regarding the physical model, it is remarkable that the time consumption is less than one milisecond, despite the high amount of collision spheres. In this aspect, we have to consider the fact that there are more patches which have to evaluate the physical model of their blades of grass, than there are visible ones. With respect to this fact, the average time consumption of the force update of a single patch is even lower compared to the visibility update. When the collision detection is disabled, the time required for the evaluation of the physical model is only one tenth of the time consumption when collision detection is enabled.

In case of all features being disabled, the frame rate drops immediately. This shows that our culling methods have strong influence on the performance of our rendering technique.

	#1	#2	#3	#4	#5	#6	#7	#8
FPS	123	119	120	119	120	107	129	**78**
T. frame	8.130	8.403	8.334	8.514	8.323	9.347	7.742	12.821
Blades drawn	43,128	49,935	48,448	**43,128**	50,244	**59,082**	43,128	168,333
Blades culled	**125,205**	118,398	119,885	125,205	118,089	109,251	125,205	0
T. update	1.948	1.994	2.197	1.957	2.069	1.822	1.433	2.426
T. force update	0.547	0.622	0.563	0.604	0.508	0.558	**0.041**	0.051
T. vis. update	1.401	1.372	**1.634**	1.353	1.361	1.464	1.392	**2.375**
T. rendering	**2.057**	2.197	2.119	2.100	2.092	2.937	2.082	**3.872**
Coll. spheres	**183**	183	183	183	183	183	0	0

Table 9.3: This table shows the evaluation of the nature scene.

9.2.2 Helicopter scene

The goal of the helicopter scene is to show the impact of the wind effect together with the rendering of a flat field of grass having high density. Apart from the field of grass, the scene only contains a complex helicopter model. The field of grass is generated with a high density value, which leads to 900,000 blades of grass divided into 59 patches. For the patch generation, the nearest-neighbor clustering approach is used. The helicopter's influence on the blades of grass is simulated by using an area wind. A rendering of this

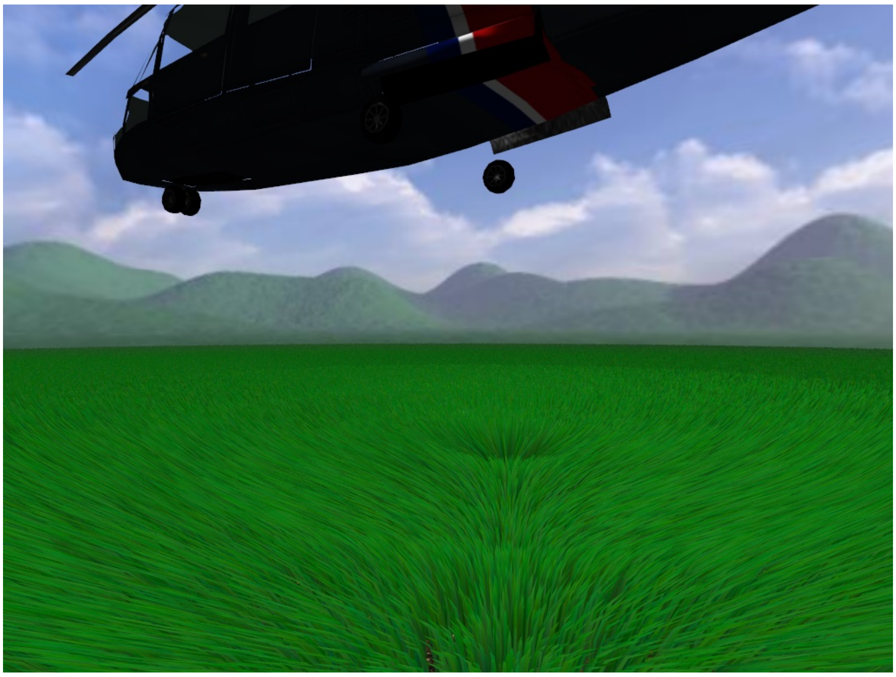

Figure 9.4: This figure presents a rendering of the helicopter scene.

scene is presented in Figure 9.4. Table 9.4 shows the results of the evaluation. Since this scene contains neither inner spheres nor collision spheres, we omit the test cases *#4* and *#7*, in which these two methods are disabled.

This test scene resembles a worst-case scenario for our rendering algorithm, since there are no occluders in this dense field of grass. Especially since the blades are seeded on a flat surface, no additional occlusion from the surface is possible. However, even if there are no occluding objects, the culling methods are able to cull about two-thirds of the blades of grass without making the field look sparse. Without occluders, the culling is only based on the camera's view frustum, the orientation of the blade and its distance to the camera.

Since the field of grass is extremely large, the method that culls most blades of grass is the distance-culling method. In our measurement, the distance range is divided into three different levels. A comparison of different amounts of levels is presented in Figure 9.5. This figure shows renderings using five, seven and nine distance levels. When five distance levels are used, the amount of culled blades increases to 354,663, which is an increase of almost five percent compared to culling using three distance levels. By using seven levels, the amount of culled blades is 360,171, which represents an increase of six and a half percent. Finally, using nine distance levels changes the amount of culled blades to 364,352, which is an increase of about seven and a half percent. This shows that the amount of culled blades of the distance-culling method is not linearly dependent on the

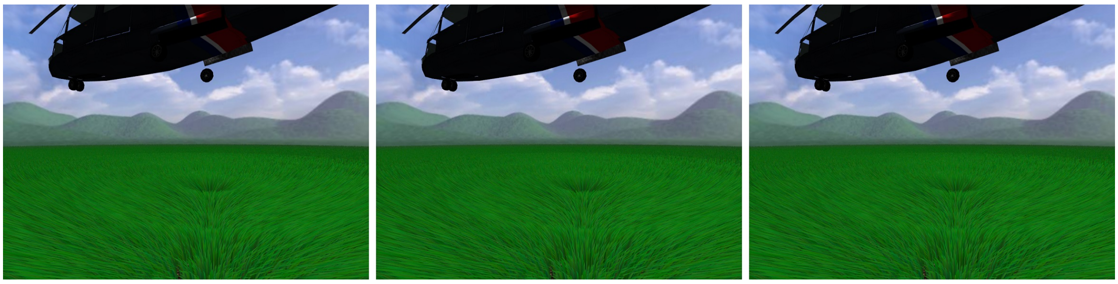

Figure 9.5: This figure shows the different amounts of distance levels. The left image shows a rendering using five distance levels. The image in the middle presents the usage of seven distance levels for the culling. In the right image, nine distance levels are calculated. Hardly any difference can be noticed, although the left image contains about four thousand blades more than the right image.

used distance levels. When a large amount of distance levels is reached, the number of culled blades even decreases by adding an additional distance level. Despite of the huge difference of culled blades, hardly any differences can be noticed in the renderings of the scene using different amounts of distance levels. This is a major advantage of the distance culling, since fewer blades have to be drawn for the same visual appearance.

Contrary to distance culling, the view-frustum culling of single blades of grass culls the smallest amount of blades. This is an indicator for the compactness of the patches. In this case, the amount of culled blades of the low-level view-frustum culling represents the amount of blades outside the view-frustum, although the respective patches intersect the view frustum.

The fifth test shows the overhead of the depth-buffer culling method. Since no blades of grass are occluded by any other object, the number of culled blades stays the same. The increase of performance is clearly visible. However, in normal scenarios there are hardly any situations where no blade of grass is hidden behind a non-transparent object.

Finally, the last test shows again the drastical increase of the running time in case of no features being enabled. As stated in the previous section, the visibility update is responsible for writing into the index buffer and for counting the amount of blades. This explains the increase of the time consumption, although no culling methods are executed.

9.2.3 Shape scene

The shape scene is created to show the ability of our analytic functions, which form the shape of a blade of grass. Therefore, this scene shows a flat field of grass on which blades of five different types and distributions are generated. All five distributions together generate 55,575 blades of grass divided into 5 patches. The blades of four of five distributions are generated using the tuft-seeding approach, which gives a heterogenous look to the field of grass. Since this scene contains no other objects than the field of

	#1	#2	#3	#4	#5	#6	#7	#8
FPS	**56**	45	46	-	**58**	43	-	**35**
T. frame	17.860	22.427	21.836	-	17.217	23.452	-	28.624
Blades drawn	165,135	**233,956**	**169,580**	-	165,135	226,425	-	503,382
Blades culled	**338,247**	269,426	333,802	-	338,247	276,957	-	0
T. update	8.239	8.324	8.687	-	8.052	8.353	-	9.712
T. force update	**1.421**	1.512	1.382	-	1.407	1.359	-	1.570
T. vis. update	6.817	6.812	7.305	-	6.645	6.994	-	**8.142**
T. rendering	5.471	6.279	5.524	-	5.535	6.669	-	**9.149**
Coll. spheres	0	0	0	-	0	0	-	0

Table 9.4: This table shows the evaluation of the helicopter scene.

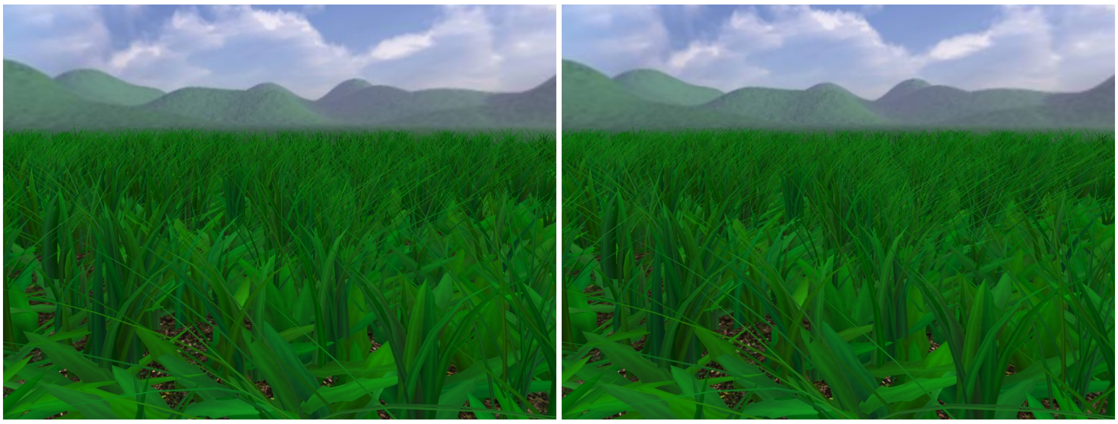

Figure 9.6: This figure presents two renderings of the shape scene. In the left image, all culling features are enabled and 12,951 blades of grass are drawn. In the right image, all culling features are disabled and 55,575 blades of grass are drawn. There is hardly any difference between the two images, although the right image renders five times as many blades as the left one.

grass, the test cases *#4*, *#5* and *#7* are omitted. A rendering of the scene is shown in Figure 9.6, and the measurements of the test cases are presented in Table 9.5.

The measurements of this scene show two major facts. First, the field of grass looks very dense and heterogenous, although it contains only a fraction of the blades of grass of the previous scenes. This is achieved by shapes that have a wide blade, like the dandelion shape, for example. In addition, the effects of the tuft-seeding approach are clearly visible, and the field has a very natural appearance.

The second fact is shown by the measured numbers. In this scene, the performance increases, although the acceleration methods are turned off. Even in the case where no blade is culled, the rendering time does not change much compared to the previous sections. This leads to the assumption that our grass-rendering technique is not limited by the graphics card if the number of blades of grass is not large, which is an indicator for the fast update and rendering mechanism of our approach.

On the right side of Figure 9.6, the scene is drawn with all 55,575 blades of grass. In comparison to the rendering on the left side of this figure, there is hardly any noticable difference in the density of the field of grass. Only a few thin blades become visible, although the amount of rendered blades is almost five times higher than in the left image.

	#1	#2	#3	#4	#5	#6	#7	#8
FPS	**170**	177	**178**	-	-	**178**	-	**164**
T. frame	5.881	5.651	5.614	-	-	5.671	-	6.110
Blades drawn	**12,951**	12,954	13,791	-	-	**17,671**	-	**55,575**
Blades culled	42,624	42,621	41,784	-	-	37,904	-	0
T. update	1.031	0.974	1.061	-	-	1.059	-	1.064
T. force update	0.145	0.130	0.119	-	-	0.125	-	0.124
T. vis. update	0.886	0.843	0.942	-	-	0.934	-	0.940
T. rendering	1.982	1.961	1.964	-	-	**2.277**	-	2.396
Coll. spheres	0	0	0	-	-	0	-	0

Table 9.5: This table shows the evaluation of the shape scene.

9.3 Discussion

The results above show that our rendering technique is capable of drawing large fields of grass of with high detail. The culling methods are essential for the performance of our algorithm. In a normal scenario, these methods are able to cull up to three-fourths of all blades of grass without drastically changing the density of the field of grass that is rendered. This is caused by three characteristics of the blades of grass. First, the blade is culled if it is not visible in the final rendering. This is the case if the blades are outside of the camera's view frustum or hidden behind the terrain and other occluding objects. Second, the blades are culled because they are far away and share the same pixels with other blades, which can cause artifacts during the rendering process. Third, blades are culled if they are viewed from the side, which makes them nearly invisible due to the two-dimensionality of our representation of a blade of grass. Another fact that can be derived from the results is that the physical model can be calculated very accurately and fast. Even for a high number of collision spheres, the total time required for the update of the physical model is only a fraction of the time needed for the rendering.

In the following sections, we compare our rendering technique to the results of related work. Since the performance aspects are difficult to compare due to different hardware and applications, we focus on the comparison between the features of the rendering technique and the visual quality. All renderings of our technique of the following sections are calculated in less than 16 miliseconds, which means at least 60 frames per second.

9.3.1 Physical interaction

The work of Orthmann et al. [JSK09] as well as the work of Fan et al. [FLHS15] focus on the physical interaction of the grass representation. Orthmann et al. use billboards for the grass representation. The physical interaction between the billboards and the environment is implemented on the graphics card. When a collision is detected, the vertices of the billboard are displaced. After a set time, the billboard regains its original state again. The physical model of this approach is capable of detecting collisions between billboards and a complex objects by using a hierarchy of spatial data structures. The algorithm of Fan et al. follows a similar procedure. In this technique, the blades of grass are represented as three-dimensional objects. The physical interaction is implemented using techniques used for fur rendering, which are implemented on the graphics card. When a collision occurs, the vertices of the blades of grass are translated. After a fixed time, this translation is nullified and the blade resets to its original position. The authors do not mention the type of the objects that are able to collide with the blades of grass. However, since only balls are shown in the results, it can be assumed that this approach is limited to the collision with spheres.

In contrast to these approaches, our technique is able to operate on each single blade of grass and saves the current physical state for each of these blades. This enables that each blade can decide on itself how much time it requires to recover from a collision. Figure 9.7 shows the comparison between our rendering technique and the approach by Orthmann et al. In our case, we used a three-dimensional model of a hand to show the interaction between a complex object and the field of grass. The rendering by Orthmann et al. shows a person that is moved through the field of grass. This shows a drawback of using billboards, because the textures appear to be heavily distorted at the billboards that are pushed down. Since the blades of grass are represented as geometric objects in our approach, the blades being pushed down have the same detailed appearance as the other blades.

In Figure 9.8, we compare our collision detection with the algorithm by Fan et al. For this comparison we generated over one hundred balls of different types, which are rolling over the scene. The rendering of the work of Fan et al. presents a similar image, although the field of grass is clearly less dense and hardly any trails comming from collisions can be noticed. This might be due to the fact that the collision is not stored for each blade of grass. Instead, it is stored per patch, which leads to less accurate collision results for a single blade of grass compared to our algorithm.

9.3.2 Wind

Almost all grass-rendering techniques have their own simulation of wind effects. In most cases, wind is represented by sine and cosine waves that are moving over the field of grass. In this section, we compare our results to those of the paper by Wang et al. [WWZ$^+$05]. In this paper the authors propose special variants of wind influence. On the one hand, they modeled a helicopter scene, similar to our test scene, and on the other hand, they

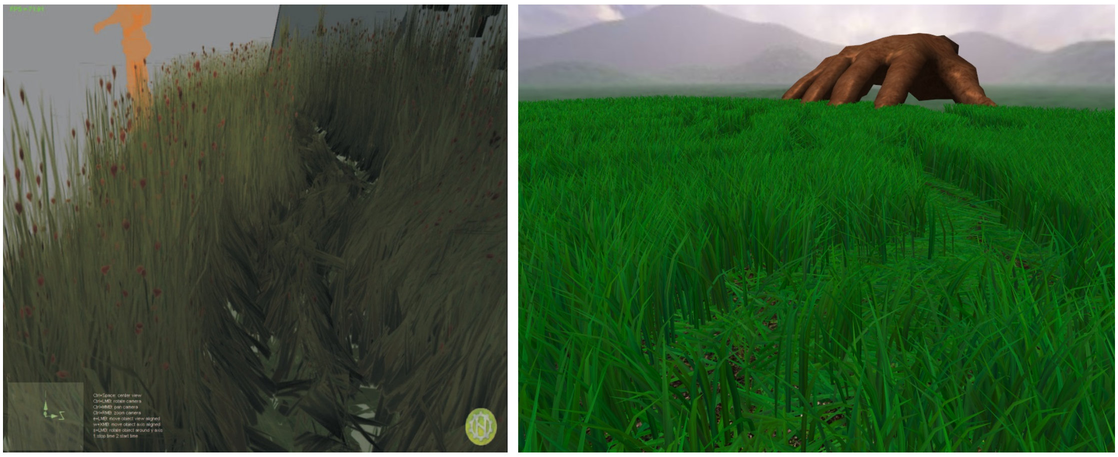

Figure 9.7: This figure shows the comparison between our technique and the approach by Orthmann et al. [JSK09]. The left image shows a person being moved through the field of grass. This image is taken from the paper by Orthmann et al. The right image presents a rendering of our technique, where we used a model of a hand to show the interaction between the field of grass and a complex object.

generate a circulating wind looking like a little tornado. Both scenes render only sparse fields of grass, which is caused by the fact that the rendering technique draws each blade of grass as single three-dimensional object. Figure 9.9 shows our implementation of the helicopter and the rotational wind compared to the renderings by Wang et al.

9.3.3 Blade geometry

Regarding the geometry of a blade of grass, there are two possible geometrical representations. Either the blade is shaped with the geometry or the blade's shape is a rectangle and the shape is formed by an alpha texture. The rendering technique by Jahrmann et al. [JW13] uses rectangular blades of grass, where the final shape is generated in the fragment shader by discarding invisible fragments based on an alpha texture. The advantage of using alpha textures is that the shape can be easily replaced by simply drawing another texture. However, the quality of an alpha texture depends on the resolution of the texture. Thus, the quality of the technique is proportional to the memory consumption. In addition, there can be sampling artifacts due to texture filtering, when the camera is near the blade of grass or far away.

In our rendering technique, the geometry of the blade of grass is adapted to the shape of the blade. Therefore, no additional texture is needed. In comparison to using an alpha texture, this approach makes it harder to introduce new shapes, since an appropriate analytic function has to be found first. However, analytic functions have many advantages. Since the geometry is adapted to the shape of the blades, no unnecessary fragments have to be calculated. In addition, the shape of the blade of grass always has the same accuracy

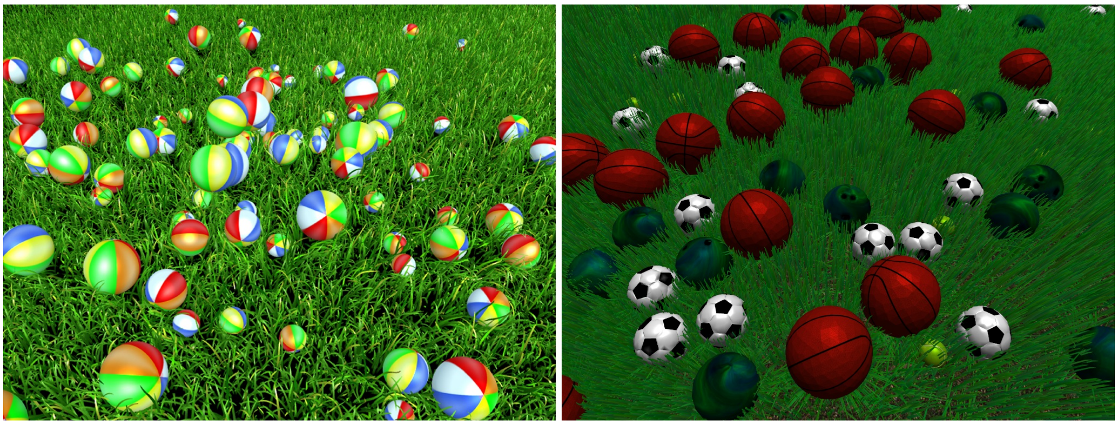

Figure 9.8: This figure shows the comparison between our technique and the approach by Fan et al. [FLHS15]. The left image shows a field of grass with many colored balls. This image is taken from the paper by Fan et al. The right image presents a rendering of our technique, including also over hundred balls being thrown over the field of grass.

regardless of the position of the camera. Figure 9.10 shows the difference between the approach of Jahrmann et al. and our rendering technique by having a detailed view of the single blades of the field of grass. In addition, this figure shows different shapes that are possible when using analytic functions.

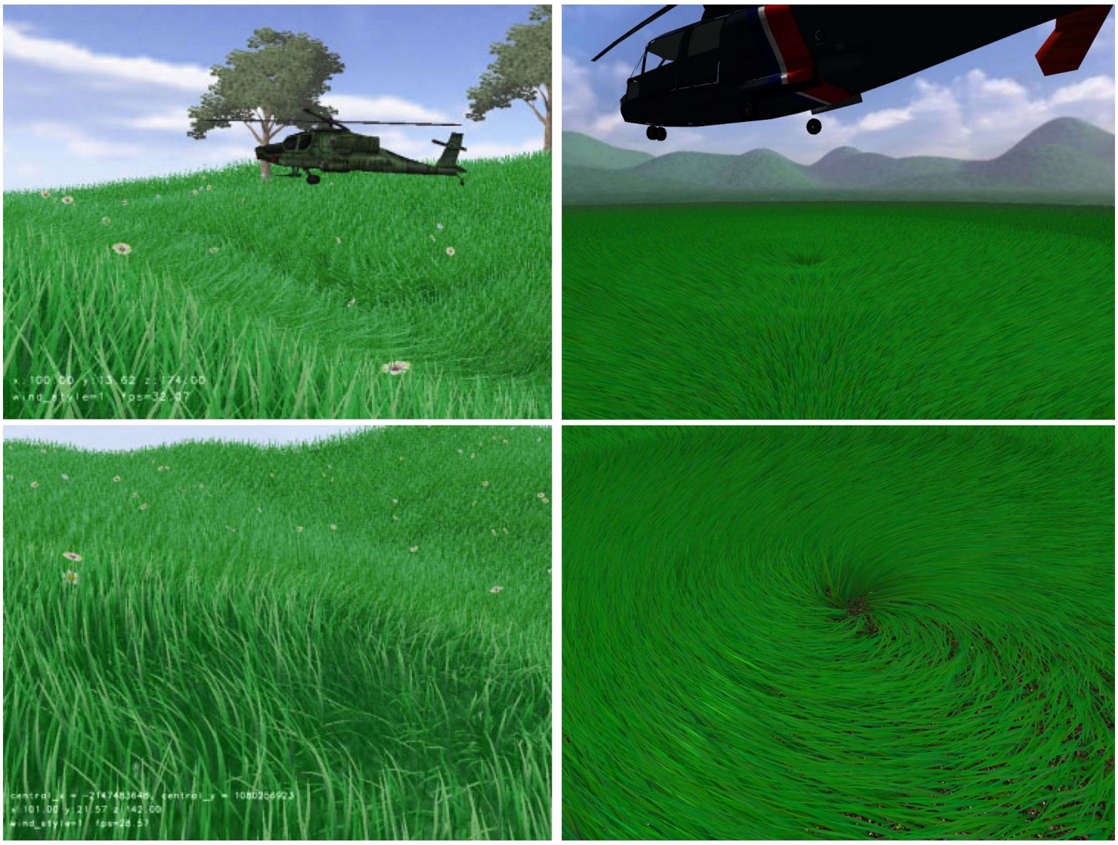
x:100.00 y:13.62 z:174.00
wind_style=1 fps=32.07
central_x = -2147483648, central_y = 1080256923
x:101.00 y:21.57 z:142.00
wind_style=1 fps=28.57

Figure 9.9: This figure shows the comparison between our technique and the approach by Wang et al. [WWZ$^+$05]. The images on the left side are taken from the paper by Wang et al. and show the different wind influences. The images on the right present renderings of our technique, where we have modeled the wind influences as stated in Chapter 5.

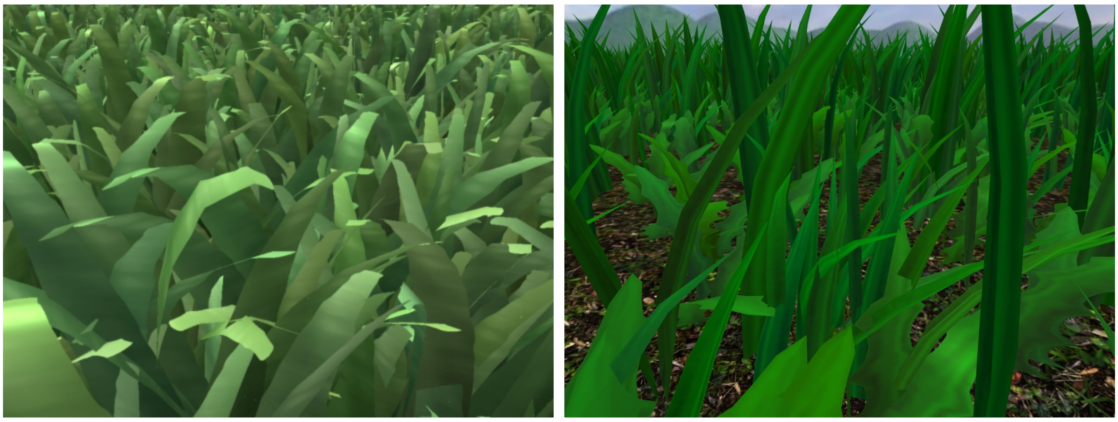

Figure 9.10: This figure shows the comparison between our technique and the approach by Jahrmann et al. [JW13]. The image on the left side is rendered by the approach by Jahrmann et al. and shows a detailed view of single blades of grass. The image on the right presents a close-up image of single blades of grass of our rendering technique having different shapes.

CHAPTER 10

Conclusion and Future Work

In this thesis, we have proposed a novel grass-rendering technique that is capable of rendering dense fields of grass in real time. In comparison to related work, the field of grass can have any shape or spatial alignment and does not require to be placed on a flat surface or a heightmap. In addition, our approach is able to render grass that has a realistic appearance and can react to its environment. This reaction to its environment is an essential contribution of our approach, which is built on a physically based model. This model includes the influence of gravity, wind, and collisions with both simple and complex objects. For the representation of complex objects, we use a sphere-packing approach that aims at generating non-overlapping spheres until the whole volume of a model is filled.

In order to achieve interactive frame rates, we use two different acceleration techniques. For the first one, we introduce a patch structure, which functions as a container for single blades of grass. Each patch contains a bounding box, which is tested for visibility before each single blade of grass is processed. The generation of the patches from single blades can be represented as a balanced clustering problem. In this thesis, we propose an accurate algorithm to solve the balanced clustering problem. However, this approach needs much time to generate the patches. Therefore, we also present some fast ad-hoc algorithms, which can generate clusters very efficiently at the cost of less optimal clusters.

Apart from the bounding-box visibility test of patches, our rendering technique is capable of evaluating the visibility of each blade of grass seperately. To this end, we introduce accurate culling methods, which are executed directly on the graphics card. By using OpenGL's indirect drawing, we are able to remove blades of grass directly from the rendering pipeline, after they are evaluated as being hidden. In addition to occluded blades of grass, the culling methods are also capable of removing blades that could cause rendering artifacts. For example, when rendering blades of grass that are far away, many blades can be projected onto the same pixel, which can generate flickering pixel artifacts, called z-fighting. Our results show that in a common view of a field of grass, almost three-fourths of the blades of grass can be culled without noticing any difference of the density of the field of grass. During the rendering of the field of grass, each single blade is drawn as a tessellated geometrical object. The shape of the blade of grass is defined by an analytic function. This allows the blade of grass to have smooth edges regardless of the distance to the camera.

In the following, we discuss some possible improvements to our grass-rendering technique, which will help to increase the performance or will introduce more realistic features.

Regarding the patch generation, more sophisticated algorithms can be implemented to achieve a lower mean square distance for the blades of grass in shorter time. For example, a linear programming solution, which minimizes the mean square distance, could be an appropriate improvement. Another alternative would be the implementation of a balanced clustering algorithm that uses graph-theory techniques, like the balanced k-means by Malinen et al. [MF14].

For the collision with complex objects, different algorithms for generating a sphere representation can be implemented. As stated in chapter 5, the spheres do not have to be packed without overlap into the three-dimensional model, as it is done in our application. More important is a good representation of the surface of the model by using spheres that are big enough to have influence on a blade of grass. For example, Stolpner et al. [SKS12] present an approach to generate an accurate sphere representation of an object, which aims at finding a good approximation of the shape of an object with a mimum amount of spheres. This could be a reasonable improvement to our sphere-packing approach, since fewer and bigger spheres can be generated, if overlapping is allowed.

As an improvement to lower the rendering time, additional level-of-detail (LOD) representations of the field of grass can be implemented, like it is done in the work of Boulanger et al. [BPB09]. The problem of different static LOD representations is to hide the transition between the different levels. The authors propose a linear blending between the different levels. However, this might be problematic for our approach, since the low-detail levels need to have the same physical state as the high-detail level. Otherwise, the transition between the different levels can easily be noticed by the physical interaction of the grass.

In addition, it would be possible to implement different effects to increase the realism of the field of grass. Some ideas for possible effects are presented below.

- The *displanting* of single blades of grass can be introduced to the physical model, if too high forces influence a blade of grass. After the blade has been displanted, it can be removed from the patch and processed like a particle in a particle system. The empty space in the patch can be filled by generating a new blade of grass, which grows by increasing its height value over time. The problem of displanting is that the patches cannot be referenced more than once, if the ground position of a blade of grass is able to change.
- A particle system can be added in order to simulate *rain*. By using well-suited bounding volumes, the rain particles can be added to the physical model, such that

they provide an additional force after colliding with a blade of grass. Furthermore, the particle can be grouped with the blade of grass after the collision. The grouping can allow the rain particle to move along the curve of the blade, which can simulate a natural rain drop on a blade of grass.

- In nature, a blade of grass is not fully opaque. Therefore, the shading model can be extended to introduce the indirect lighting comming from the *translucency* from neighboring blades of grass.
- Additionally, *shadows* and *ambient occlusion* can be introduced into the grass-rendering technique, to increase the visual depth of the grass representation. The problem for these techniques is the high amount of geometry, which can cause shading artifacts if the sampling of the used techniques is not good enough.

List of Figures

2.1 This figure shows images of three different image-based grass rendering techniques. a) represents a rendering from the technique proposed by Neyret et al. [Ney95], b) is extracted from the paper by Shah et al. [SKP05] and c) gives an example showing the result of the algorithm of Habel et al. [HWJ07]. . . . 6
2.2 This figure shows renderings of three different non-interactive billboard techniques. a) represents an image from the GPU gems article [Wha05], b) is taken from the paper by Perbet et al. [PC01] and c) shows a rendering from the GPU gems article [Pel04]. 6
2.3 This figure shows examples of two interactive billboard techniques. a) and b) are extracted from the paper by Orthmann et al. [JSK09] and c) represents a rendering from the algorithm by Chen et al. [CJ10]. 7
2.4 This figure shows examples of two geometrical approaches. a) is taken from the paper by Wang et al. [WWZ$^+$05] and b) represents a rendering from the algorithm by Zhao et al. [ZLZ09]. 8
2.5 This figure shows renderings of the two most modern geometrical approaches. a) represents a rendering taken from the paper by Jahrmann et al. [JW13] and b) shows an image from the algorithm by Fan et al. [FLHS15]. 9
2.6 This figure shows examples of the hybrid approach by Boulanger et al. [BPB09]. a) represents a visualization of the different static level-of-detail stages and b) shows a rendering from the proposed algorithm. Both figures are extracted from the paper by Boulanger et al. [BPB09]. 10

3.1 This figure shows a sketch of the levels of the hierarchy. 13
3.2 This figure illustrates the steps of the basic algorithm. 13

4.1 This figure outlines the definition of a blade of grass. The Bézier curve is shown together with its three control points. Additionally, all attributes apart from the bending value are presented together with their impact on the blade of grass. 16
4.2 This figure presents the scheme of the algorithm for distributing blades of grass on the surface of a three-dimensional model. 17

4.3 This figure presents the comparison of different spatial distribution in combination with different blade generation algorithms. The rows represent the blade generation algorithms: the upper row refers to the single-blade seeding and the lower row to the tuft seeding. The columns represent the spatial distribution algorithms: the left column refers to the random-face distribution, the middle column to the area distribution and the right column to the octree distribution. The *bend* parameter of the blades of grass is set very low, such that the positioning of the blades is more visible. The three-dimensional model is taken from the Standford scanning repository [Lab]. 19

4.4 This figure compares the distribution betwen the area distribution (upper image) and the volumetric distribution (lower image) for a special case. The red lines indicate the generated blades of grass and the green dots represent the randomly sampled points of each distribution. Both images contains the same amount of blades and sample points. 21

4.5 This figure compares the two different possibilities of blade generation. The left image shows the single-blade seeding and the right image the tuft seeding. The fields of grass of both images have the same amount of blades. 22

5.1 This figure shows an illustration of the physical model. The vectors represent the different forces that are taken into account to translate the control point $\mathbf{v}_2$. The sphere indicates a collider, and the vector on top shows the current direction of the wind. 32

5.2 This figure shows an example of a force map. Black pixels either refer to only negative displacements or to unused pixels. Thus, the amount of black pixels can be an indicator on the quality of the distribution of the patches. Most of the tiles refer to a meadow aligned along the xz-plane. Therefore, there will be no green displacement of a blade of grass, since this would mean that a blade is torn off the ground by a force. The upper five tiles refer to grass patches on a bunny model. In these tiles all different colors are represented. This can be explained by the blades of grass having a higher variety in their up-vectors, which can lead to displacements in each direction. 34

5.3 This figure illustrates the directional wind influence in two-dimensions. The height of the red surface indicates the influence $w_{\mathbf{p}}\left(\vec{\lambda}\right)$ for a specific position. If the surface is high, the wind has strong influence on the respective position. The right image shows the position-related influence with a higher value of $\vec{\lambda}_w$. The black arrow represents the movement of the wind wave for a better understanding. 37

5.4 This figure illustrates the area wind influence in two-dimensions. The height of the red surface indicates the strength of the influence $w_{\mathbf{p}}\left(\vec{\lambda}\right)$ for a specific position. The position of the wind source is located in the center of the visualization. The right image shows the position-related influence with a higher value of $\vec{\lambda}_w$. The black arrows represent the movement of the wind wave. 38

5.5 This figure illustrates the relation between $\mathbf{v}_1$ and $\mathbf{v}_2$. The different colors symbolize different states of the blade of grass. 41

5.6 This figure illustrates two cases of the collision between a sphere and a blade of grass. In image a) $\mathbf{v}_2$ is inside of the sphere and is moved to the boundary of the sphere. The resulting translation is indicated by the red vector. In image b) the mid point is inside of the sphere and has to be moved to its boundary. Since the physical model modifies $\mathbf{v}_2$ only, $\mathbf{v}_2$ is translated fourfold by the translation of the mid point. These translations are indicated by the two parallel red vectors. 43

5.7 This figure shows three examples of sphere-packed objects. The models are taken from the Standford scanning repository [Lab]. 46

6.1 This figure shows the artifact that occurs if the force update is not calculated outside of the view frustum. The rendered image is taken after a quick turn of the camera and the scale-formed patch borders can be seen clearly. 49

6.2 This figure illustrates the different levels of the distance culling method. The red color indicates the lowest level whereas the green color represents the highest level. 54

6.3 This figure shows the accuracy of the occlusion-culling methods. The top row presents examples of inner-sphere culling, whereas the bottom row illustrates depth-buffer culling. The images of the left column show the scenes rendered normally. The middle images show renderings where the wireframe mode is enabled. Finally, the images of the right column present the renderings without culling. 55

7.1 This figure shows a comparison between the rendering algorithm proposed by Jahrmann et al. [JW13] and the rendering algorithm presented in this thesis. The top row is extracted from the paper. The first and the second step represent the in- and output of the vertex shader. The third and fourth image shows the input and the result of the tessellation evaluation shader and the last picture outlines a shaded blade of grass. The lower row represents the proposed algorithm. In this row, the first image, showing three points, represent both the in- and the output of the vertex shader. The second to the fourth picture shows the input, the intermediate result before shaping and the output of the tessellation evaluation shader. Finally, the last image outlines a shaded blade of grass of this method. 58

7.2 This figure is taken from the paper by Jahrmann et al. [JW13]. Image a) shows an illustration of De Casteljau's algorithm for calculating a point on a curve with the parameter v. Image b) presents the calculation of the tangent and bitangent of the blade of grass. 60

7.3 This figure shows the four basic shapes that can be applied to a blade of grass. The axes represent the u and v parameters and the dotted red and green lines indicate the positions of $\mathbf{c}_1$ and $\mathbf{c}_2$. The parameter u can have the distinct values 0, 0.5 and 1, whereas v can have any value in the interval $[0, 1]$ depending on the tessellation level. 62

7.4 This figure shows the graphs of three different levels of the dandelion function. The blue graph corresponds to level 1, the red one to level 2 and the green one to level 3. 64

7.5 This figure shows the different results whether the flare effect is enabled or disabled. The left image shows a frame rendered without flare effect, whereas the right frame is rendered with the flare effect enabled. 66

9.1 This figure shows the resulting patch divisions of the different patch-generation algorithms of the flat-surface division. The topmost image is generated by using the largest-dimension sorting algorithm. Regarding the images below, the columns present the methods as follows: the seed-point algorithm using the box-subdivision method, the seed-point algorithm using the dimension-factorization method and the clustering algorithms. The rows for the first two columns represent the normal seed-point algorithm using the Euclidean and the Manhattan distance metric. Below, the images show the advanced seed-point algorithm using the Euclidean and the Manhattan distance metric. The last column shows the different variants of the clustering algorithms. The first two rows represent the nearest-neighbor clustering algorithm using the Euclidean and the Manhattan distance metric. The last two rows present the results of the k-means clustering algorithm using the Euclidean and the Manhattan distance metric. 79

9.2 This figure shows the resulting patch divisions of the different patch-generation algorithms of the complex-model division. The topmost image is generated by using the largest-dimension sorting algorithm. Regarding the images below, the columns present the methods as follows: the seed-point algorithm using the box-subdivision method, the seed-point algorithm using the dimension-factorization method and the clustering algorithms. The rows for the first two columns represent the normal seed-point algorithm using the Euclidean and the Manhattan distance metric. Below, the images show the advanced seed-point algorithm using the Euclidean and the Manhattan distance metric. The last column shows the different variants of the clustering algorithms. The first two rows represent the nearest-neighbor clustering algorithm using the Euclidean and the Manhattan distance metric. The last two rows present the results of the k-means clustering algorithm using the Euclidean and the Manhattan distance metric. 81

9.3 This figure shows renderings of the nature scene. The left image shows a normal view of the scene. The middle image presents the collision spheres of the bunny model, which are generated by the sphere packing algorithm. The right image represents a wire-frame view of the scene. It can be seen that the tree is too thin to completely hide blades behind it. However, the culling of the other objects can be noticed easily in the this rendering mode. 84

9.4 This figure presents a rendering of the helicopter scene. 86

9.5 This figure shows the different amounts of distance levels. The left image shows a rendering using five distance levels. The image in the middle presents the usage of seven distance levels for the culling. In the right image, nine distance levels are calculated. Hardly any difference can be noticed, although the left image contains about four thousand blades more than the right image. 87

9.6 This figure presents two renderings of the shape scene. In the left image, all culling features are enabled and 12,951 blades of grass are drawn. In the right image, all culling features are disabled and 55,575 blades of grass are drawn. There is hardly any difference between the two images, although the right image renders five times as many blades as the left one. 88

9.7 This figure shows the comparison between our technique and the approach by Orthmann et al. [JSK09]. The left image shows a person being moved through the field of grass. This image is taken from the paper by Orthmann et al. The right image presents a rendering of our technique, where we used a model of a hand to show the interaction between the field of grass and a complex object. 91

9.8 This figure shows the comparison between our technique and the approach by Fan et al. [FLHS15]. The left image shows a field of grass with many colored balls. This image is taken from the paper by Fan et al. The right image presents a rendering of our technique, including also over hundred balls being thrown over the field of grass. 92

9.9 This figure shows the comparison between our technique and the approach by Wang et al. [WWZ$^+$05]. The images on the left side are taken from the paper by Wang et al. and show the different wind influences. The images on the right present renderings of our technique, where we have modeled the wind influences as stated in Chapter 5. 93

9.10 This figure shows the comparison between our technique and the approach by Jahrmann et al. [JW13]. The image on the left side is rendered by the approach by Jahrmann et al. and shows a detailed view of single blades of grass. The image on the right presents a close-up image of single blades of grass of our rendering technique having different shapes. 94

List of Tables

9.1 This table represents the evaluation of the flat-surface division. The running time of the generation of a field of grass using different patch generation algorithms is measured. Likewise, the compactness of the patches is quantified by the minimum square distance of the resulting patches. 80
9.2 This table represents the evaluation of the complex-model division. The running time of the generation of a field of grass using different patch generation algorithms is measured. Likewise, the compactness of the patches is quantified by the minimum square distance of the resulting patches. 82
9.3 This table shows the evaluation of the nature scene. 85
9.4 This table shows the evaluation of the helicopter scene. 88
9.5 This table shows the evaluation of the shape scene. 89

List of Algorithms

4.1 Distribution - Face area 20
4.2 Distribution - Octree 22
4.3 Cluster optimization algorithm 27
4.4 Nearest-neighbor algorithm 28
4.5 Factorize 3 30

6.1 Indirect rendering 50
6.2 Depth-texture culling 56

8.1 Maximum inner sphere - optimization algorithm 75

Bibliography

[AAK+09] O. Aichholzer, F. Aurenhammer, B. Kornberger, S. Plantinga, G. Rote, A. Sturm, and G. Vegter. Recovering structure from r-sampled objects. *Computer Graphics Forum*, 28(5):1349–1360, 2009.

[BDM09] Rainer E Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems, Revised Reprint.* Siam, 2009.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[Ber06] P. Berkhin. *A Survey of Clustering Data Mining Techniques*, pages 25–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[BPB09] Kévin Boulanger, Sumanta N. Pattanaik, and Kadi Bouatouch. Rendering grass in real time with dynamic lighting. *IEEE Comput. Graph. Appl.*, 29(1):32–41, January 2009.

[CJ10] K. Chen and H. Johan. Real-time continuum grass. In *2010 IEEE Virtual Reality Conference (VR)*, pages 227–234, March 2010.

[ERC01] Cass Everitt, Ashu Rege, and Cem Cebenoyan. Hardware shadow mapping. *White paper, nVIDIA*, 2, 2001.

[FH00] Gerald E Farin and Dianne Hansford. *The essentials of CAGD.* AK Peters Natick, 2000.

[FLHS15] Zengzhi Fan, Hongwei Li, Karl Hillesland, and Bin Sheng. Simulation and rendering for millions of grass blades. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, i3D '15, pages 55–60, New York, NY, USA, 2015. ACM.

[Fol96] James D Foley. *Computer Graphics: Principles and Practice*, volume 12110. Addison-Wesley Professional, 1996.

[FotUN] Food and Agriculture Organization of the United Nations. Are grasslands under threat? `http://www.fao.org/ag/agp/agpc/doc/grass_stats/grass-stats.htm`. Accessed September 10, 2016.

[Gra93] Jens Gravesen. *Adaptive subdivision and the length of Bezier curves.* Mathematical Institute, Technical University of Denmark, 1993.

[Har75] John A Hartigan. *Clustering algorithms.* 1975.

[HM09] Mhand Hifi and Rym M'hallah. A literature review on circle and sphere packing problems: models and methodologies. *Advances in Operations Research*, 2009, 2009.

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

[JSK09] Orthman Jens, Christof Rezk Salama, and Andreas Kolb. Gpu-based responsive grass. 2009.

[JW13] Klemens Jahrmann and Michael Wimmer. Interactive grass rendering using real-time tessellation. In Manuel Oliveira and Vaclav Skala, editors, *WSCG 2013 Full Paper Proceedings*, pages 114–122, June 2013.

[Lab] Stanford Computer Graphics Laboratory. The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`. Accessed September 30, 2016.

[McM03] Alison McMahan. Immersion, engagement and presence. *The video game theory reader*, 67:86, 2003.

[MF14] Mikko I. Malinen and Pasi Fränti. *Balanced K-Means for Clustering*, pages 32–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[Mit87] Don P. Mitchell. Generating antialiased images at low sampling densities. *SIGGRAPH Comput. Graph.*, 21(4):65–72, August 1987.

[Ney95] Fabrice Neyret. A General and Multiscale Model for Volumetric Textures. In *Graphics Interface*, pages 83–91, Toronto, Canada, 1995.

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

[Pel04] Kurt Pelzer. Rendering countless blades of waving grass. In Randima Fernando, editor, *GPU Gems*, pages 107–121. Addison-Wesley, 2004.

[Pow91] David M W Powers. Parallelized quicksort and radixsort with optimal speedup. In *PROCEEDINGS OF INTERNATIONAL CONFERENCE ON PARALLEL COMPUTING TECHNOLOGIES. NOVOSIBIRSK.*, pages 167–176. World Scientific, 1991.

[SA16] Mark Segal and Kurt Akeley. The OpenGL® Graphics System: A Specification. `https://www.opengl.org/registry/doc/glspec45.core.pdf`, July 2016. Accessed September 30, 2016.

[Sco01] Jim Scott. Point in triangle test. `http://blackpawn.com/texts/pointinpoly/`, November 2001. Accessed September 30, 2016.

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

[SKS12] S. Stolpner, P. Kry, and K. Siddiqi. Medial spheres for shape approximation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(6):1234–1240, June 2012.

[Ste13] Wesley Stevenson. Spatial clustering with equal sizes. `http://statistical-research.com/spatial-clustering-with-equal-sizes/`, November 2013. Accessed September 16, 2016.

[Wha05] David Whatley. Toward photorealism in virtual botany. In Matt Pharr, editor, *GPU Gems 2*, pages 7–25. Addison-Wesley, 2005.

[WWZ$^+$05] Changbo Wang, Zhangye Wang, Qi Zhou, Chengfang Song, Yu Guan, and Qunsheng Peng. Dynamic modeling and rendering of grass wagging in wind: Natural phenomena and special effects. *Comput. Animat. Virtual Worlds*, 16(3-4):377–389, July 2005.

[WZ10] Rene Weller and Gabriel Zachmann. Protosphere: A gpu-assisted prototype guided sphere packing algorithm for arbitrary objects. In *ACM SIGGRAPH ASIA 2010 Sketches*, SA '10, pages 8:1–8:2, New York, NY, USA, 2010. ACM.

[ZLZ09] X. Zhao, F. Li, and S. Zhan. Real-time animating and rendering of large scale grass scenery on gpu. In *Information Technology and Computer Science, 2009. ITCS 2009. International Conference on*, volume 1, pages 601–604, July 2009.