GPU-Based Responsive Grass

Diploma Thesis in Computer Science

submitted by

Jens Orthmann

born on February 22, 1981 in Hachenburg, Germany

Written at

Computer Graphics and Multimedia Systems Group
Faculty 12
University of Siegen, Germany.

Advisors:

Prof. Dr. A. Kolb (University of Siegen, Computer Graphics Group)

Dr. C. Rezk-Salama (University of Siegen, Computer Graphics Group)

Started on: December 01, 2007

Finished on: April 30, 2008

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Siegen, den 01. Februar 2008

Übersicht

Große und naturgetreue Umgebungen stellen oft einen unverzichtbaren Bestandteil der heutigen Computerspiele dar. Um die Erwartungen der Spieler an lebendige Spielräume zu erfüllen und eine höhere Immersion zu bewirken wird daher viel Wert auf die Implementierung von Interaktionsmöglichkeiten mit der Spielumgebung gelegt. Für eine möglichst realitätsnahe Simulation der Grasfläche wurden bislang vor allem Ansätze bezüglich Animation und Rendering entwickelt. In diesem Zusammenhang stellt meine Diplomarbeit eine effiziente Methode zur Simulation von deformierbarem Gras vor, die in Echtzeit auf moderner Graphikhardware umgesetzt wird. Die einzelnen Grasbüschel werden bei dieser Implementierungsstrategie in zwei unterschiedlichen kollisionsabhängig ausgewählten Typen von Gras- Billboards approximiert. Erstmalig in der neuen Graphikhardware vorhandene Stufen in der Rendering-Pipeline ermöglichen dabei eine Kollisionsbehandlung direkt auf der GPU. Die Reaktion auf Szeneobjekte erfolgt auf Basis von Distance Maps. Wird anhand der Auswertung dieser Daten eine Kollision des Szeneobjekts mit einem oder mehreren Gras-Billboards erkannt erfolgt die Verformung der betroffenen Billboards. Im Fall einer Kollision und der daraus resultierenden Verformung der Billboards wird eine unerwünschte Überdehnungen mit Hilfe von entfernungsabhängigen Federn zwischen den Vertices unterbunden. Während des darauf folgend ablaufenden Regenerationsprozesses, der im Rahmen dieser Arbeit eigens entwickelt wurde, wird die ursprüngliche Form der Billboards wieder hergestellt. Dieser Regenerationsprozess stellt eine gute Performanz sicher. Die zu rendernden Primitive des Billboards werden erst während der Renderingphase zusammengesetzt. Ein auf Ambient-Occlusion basierendes Irradiance-Volumen ermöglicht die dynamische Beleuchtung der Vertices. Das letztendliche Erscheinungsbild der Gras-Ebene wird schließlich anhand des Blendings auf Basis von Alpha-to-Coverage generiert. Abgesehen von der Vorstellung der theoretischen Konzepte, die diesen Techniken zugrunde liegen, wird im Rahmen dieser Ausarbeitung abschließend auch die Performanz der auf der GPU stattfindenden Prozesse besprochen.

Abstract

Often large natural environments are essential for todays computer games. Interaction with the environment is widely implemented in order to satisfy the player's expectations of a living scenery and to help to increase the immersion of the player. Therefore every effort is made towards the implementation of options for interaction with the game-environment. However, in order to achieve a grass simulation as realistic as possible mainly animation and rendering approaches for grass have been researched so far. Within this context my work describes an efficient way to simulate a responsive grass layer with todays graphics cards in real-time. Using the implementation strategie introduced by this diploma thesis clumps of grass are approximated by two billboard representations. Newly introduced stages of the rendering pipeline, first existing on the new graphics hardware, allow the collision handling to take place on the GPU. Distance maps are employed to respond to scene objects. If the analysis of the distance maps indicates a collision of the scene object with one or more gras billboards the deformation of the concerned billboard takes place. In case of collisions and the resulting deformation of the billboards, length constraints preserve the shape of deformed billboards. The recovering process developed throughout this thesis takes place after the deformation caused by colliding with the scene object and restores the original that is to say the undeformed shape for each of the billboards. Additionally, this regeneration process garantees the good overall performance. The primitives of the billboards are assembled during the rendering process. Their vertices are dynamically lit within an ambient occlusion based irradiance volume. Alpha-to-Coverage is used to create the final appearance of the grass layer. Besides the presentation of the theoretical concept that provides the basis of the above-mentioned techniques, the performance concerning the GPU based handling is discussed within the latter part of the examination thesis.

Acknowledgements

First of all, I would like to thank my supervisors Prof. Dr. Andreas Kolb and Dr. Christof Rezk-Salama of the Computer Graphics Group of the University of Siegen, Germany, for their support and their supervision of my work. It is a privilege to be educated by such experienced experts in computer graphics. Furthermore, I would like to thank Dr. Christof Rezk-Salama and Maik Keller for their always valuable advise.

I would also like to thank Mechtild Brenner and Marcel Piotraschke who offered their spare time to proof-read this paper.

Finally and above all I would like to express my deep gratitude to my parents Agnes and Anton Orthmann and my girlfriend Danica Brenner for their love and extraordinary emotional support. They helped me throughout all of the difficult time.

Jens Orthmann

Preface

This diploma thesis describes the results achieved during my diploma thesis project which was enabled by my supervisors Dr. Christof Rezk-Salama and Prof Dr. Andreas Kolb, Computer Graphics Group of the University of Siegen. The subject of my work was chosen out of personal interest and is not affiliated to any university project.

Within the project work I extended a standard approach concerning the simulation of grass utilizing research on the area of cloth simulation and hardware based collision handling to simulate responsive grass. The workload of the collision handling is shifted to the GPU in order to achieve real-time frame rates. I implemented the system onto the basis of a modern graphics engine working on DirectX 10.

The work consists of 8 chapters. The first chapter starts with an explanation of the motivation and gives a short overview of the system. Chapter 2 deals with the previous work in the field of grass simulation, collision detection and cloth simulation. The potentialities of todays GPUs are presented in brief in Chapter 3. The basic components of the grass layer are described in Chapter 4, including the animation of the grass billboards. Being the main component of this diploma thesis Chapter 5 tells how to achieve the responsiveness to dynamic objects which are moving through the grass layer. Chapter 6 presents the rendering process which utilizes a global illumination model in order to achieve a realistic shading of the grass primitives. The description of the responsive grass layer is concluded in Chapter 7 by a presentation of the visual results. In addition the performance concerning the collision system and the rendering system is analyzed. The last chapter summarizes the diploma thesis main matters and closes with a preview of future work. Finally, the appendix describes GPU-based distance maps as they are utilized throughout the system.

Abbreviations and Symbols

1D	One Dimensional
2D	Two Dimensional
3D	Three Dimensional
GPU	Graphics Processing Unit
CPU	Central Processing Unit
API	Application Programming Interface
AABB	Axis Aligned Bounding Box
MSAA	Multisample Anti-Aliasing
$\mathbf{T}$	matrix filled in column-major order
$\otimes$	component based vector multiplication
$\bullet$	scalar product between two vectors
$\times$	cross product between two vectors
$\|\mathbf{d}\|$	length of a vector
$\|a\|$	amount of scalar value
max(a, b)	maximum of a and b
min(a, b)	minimum of a and b
floor(a)	next smaller integer of a
ceil(a)	next higher integer of a
norm(x)	the normal at coordinate $\mathbf{x} \in [0, 1]^2$
dist(x)	the distance at coordinate $\mathbf{x} \in [0, 1]^2$
vol(x)	look up into a 3D volume at coordinate $\mathbf{x} \in [0, 1]^3$
env$(\mathbf{t}, dt_x, dt_y)$	derivative based sampling function at coordinate $\mathbf{t} \in [0, 1]^2$

Contents

List of Figures

List of Code Samples

Chapter 1

Introduction

State-of-the-art 3D games demonstrate the power of currently available graphics hardware for rendering exciting natural sceneries in real-time. In recent years this task has turned out to be difficult due to the huge number of plants. Therefore, most research applied to natural sceneries focused on the rendering and animation of a great number of plants (blades of grass, shrubs, trees etc.) in real-time. Static level design used in many previous implementations is more and more replaced by dynamic environments that can be modified in real-time throughout the gaming process. Due to the fact that natural behavior is better approximated in the game, the player feels a higher immersion while playing [McM03]. The more of the player's expectations are satisfied the more the realism of the scene is effected. Furthermore, the dynamic environment is becoming more and more a part of the game logic: Trees are chopped to obstruct the path, soldiers are creeping well disguised in the bushes and objects like boxes need to be moved in order to follow up the path. Following this trend, this work takes dynamic environments one step further by integrating responsive real-time simulation of grass. Besides the more natural look-and-feel, responsive grass will significantly improve the challenges in game play and tactics of modern games.

An efficient technique for rendering and animation of responsive grass is developed, which integrates well into existing game engines. The implementation targets Shader Model 4 graphics boards, including geometry shaders and stream output. Collision detection with dynamic scene objects, response and recovering are handled directly by the GPU. The system comprises the following components:

- **Procedural Generation of Billboard Sets:**
  For a given terrain mesh, billboards for grass blades are generated automatically by a geometry shader using a set of texture images which define the extent, direction of growth and the amount of randomness for the plant cover. This geometry shader is executed once for each tile of terrain, and the results are stored in local video memory using the stream-out capabilities.

- **Dynamic Objects:**
  Dynamic objects capable of colliding with the plant cover are represented by depth cube maps

for efficiency. These cube maps are computed by projecting the object's mesh onto the faces of a bounding cube. They are updated for each frame to account for animated objects. Additionally, for coarse collision tests, the objects are represented as a union of a fixed number of bounding spheres.

- **CPU Predecision:**
  At run-time a coarse pre-test for collision is performed by the CPU. The spatial distribution of the complete plant cover is represented as an octree of axis-aligned bounding boxes. The CPU checks whether or not the collider objects intersect with an octree node. According to the results of this test, detailed collision detection, reaction and recovering is performed on the GPU.

- **Collision Pass:**
  If a collision is possible, a geometry shader first performs a bounding sphere test, and eventually a detailed collision test against the cube-map representation of each colliding object. If a collision is detected, cloth simulation techniques based on spring models are employed for collision reaction.

- **Recover Pass:**
  After a collision has occurred the plant cover will smoothly recover. Therefore, a tile of billboards will stay active for a fixed amount of time after collision. A separate geometry shader moves the grass blades back to their original position. After the recover time has elapsed, the billboards will be again handled as simple quads.

- **Runtime Tessellation:**
  Depending on the outcome of the collision test, a billboard is represented by a simple quad or tessellated into a small mesh to account for possible deformations (collision or recovering phase).

- **Rendering:** To integrate ground vegetation into a dynamic global lighting environment, a precomputed irradiance volume is employed. These technique is adapted for realistic rendering of dynamic ground vegetation. To avoid expensive depth-sorting of the semi-transparent billboards, Alpha-to-Coverage allows order-independent rendering on the GPU while maintaining a consistent visual appearance.

Chapter 2

Related Work

Previous research in simulating interactive grass or plants tends to focus either on realistic rendering or on real-time animation as cited in Section 2.1. The lack of grass-interaction makes it necessary to go through a more general range. Therefore studies of hardware based collision detection are employed as revealed in Section 2.2. Furthermore, the cloth models that have influenced the design of the grass structure are outlined in Section 2.3.

2.1 Grass Simulation

In general, previous work on grass simulation deals with representation, real-time animation, and illumination aspects. All three subjects which made up the research on grass are described throughout this section.

2.1.1 Representation

As nature scenes often include a lot of plants (blades of grass, shrubs, trees etc.) the rendering of a high number of them is still challenging. Furthermore, they cannot be displayed with complex geometry in real time. Therefore two main strategies have been applied to solve the problem:

Since vegetation is visible from near to very far distances, many of the approaches make use of level of detail (LOD) techniques to preserve the real-time constraint. If plants are close to the camera, lit and shadowed geometry [GPR+03, BPB06], a 3D volume representation [PC01], or a cluster of billboards [BCF+05, FS04] are used to display them. If the distance increases, they are substituted by vertical and horizontal slices of 2D textures. Bakay et al. [BH02] manage the complexity without any LOD approach. They render displaced maps with semi-transparent shells to generate the illusion of grass. Even so it is not easy to apply collision detection or reaction in real-time on grass blades that are approximated by these volume rendering approaches. Guerraz et. al. , however, presented an approach to tread on the grass layer. A primitive is moved along the character's trajectory while affecting the procedural animation process of the grass [GPR+03]. Nevertheless there still is no possibility to react

to collision, based upon the object's geometry. Additionally, depending on the area where grass is planted, a great amount of memory is consumed by such volume rendering approaches. Hence an aperiodic tiling scheme is used to solve the problem [BCF+05, FS04, BPB06]. In such a scheme, the vegetation is arranged by randomly repeated tiles which store the required LOD data for a chunk of plants. Thus, the reuse of data amplifies the problem of collision response to a local tile.

Another approach to render complex geometry, especially grass with repetitive detail, is image based rendering. Therefore, view-aligned quads with a semi-transparent 2D texture, so called billboards, represent an amount of complex geometry [MH99]. Thus, the rendering based on images is much more efficient than using classical geometry and accordingly, the collision response is even easier to implement. Former suggestions use randomly distributed [IC02] and fixed aligned billboards [PC01] in order to approximate grass blades, but this leads to a lack of parallax effect. They are arranged view aligned [Wha05] and crossed [Pel04] in order to ensure a better volumetric illusion, depending on the line of sight. All vertices are stored in one large buffer that can be rendered in one single draw call [Wha05].

2.1.2 Animation

So far, there exist various techniques to animate grass billboards affected by wind on the GPU. Almost all of them project the grass vertices onto a two dimensional grid. Afterwards, trigonometric functions produce one or more positions depending on periodic values for each grid point. Ramraj [Ram05] even extends the model by a wave propagation model, common for water surfaces. In addition to the result determined by the function, each grid point is affected by its neighbours. The result is used to either rotate or bend the stalk of grass. In [Pel04] the time and position of the sprite's vertex are taken into account as a parameter to the trigonometric function. Afterwards the function's output is used to translate the vertex along the wind direction. Additionally a blend weight is assigned to each vertex premultiplied with the resulting bend value before the displacement in order to simulate the grass being more or less rigid [Wha05, Bot06]. A further elaboration of the approach is delivered by Tiago Sousa [Sou07]: A texture stores a bending sensitivity for each vertex. To receive the final displacement of a vertex, the shader sums up triangle waves to generate the displacement direction which finally is multiplied by the per vertex stiffness as well.

2.1.3 Rendering

The rendering of vegetation is a complex task. The research affects all aspects of physically correct illumination models namely the global illumination, the local reflection properties, and the correct simulation of the semi-transparent nature of grass as a part of the material.

2.1.3.1 Global Illumination

Dynamic and global illuminations of natural sceneries are rather difficult without even considering the great number of plants. The equation for global illumination [Kaj86, GTGB84] cannot be evaluated for complex scenes in real-time. Nonetheless, many approximation techniques yield good results.

In [BCF+05] the vegetation is lit by precomputing the radiance transfer for each plant: As spherical harmonics define an ortho-normal basis over the sphere, the rendering equation, which is parametrized over the hemisphere, can be projected onto the so called spherical harmonics near the object's surface. As a result, a transfer vector filled with lighting coefficients is determined. Such a vector on the surface defines how the surface reacts to incident light at that point and consequently can be used for fast illumination during run-time: The lighting environment is projected onto the spherical harmonics basis as well. In the diffuse case, a dot product of the two coefficient vectors results in a realistic illumination based upon the current lighting situation [SKS02]. Behrendt et. al. [BCF+05] note that natural environments are illuminated by a low-frequency lighting. Thus only two or three bands are needed to pre-process their plants, which results in less coefficients per vertex.

Moreover, an approach called ambient occlusion is used to precompute occlusion information for a static natural scene [BPB06]. ambient occlusion was first introduced by Hayden Landis [Lan02]. In a preprocessing step, an accessibility value as well as an average incident light direction is computed for each point of the model. The accessibility value describes the fraction of the hemisphere above the point which is unoccluded by other parts of the model. At runtime an environment map is sampled along the reflected average incident light direction and the resulting irradiance value then is attenuated by the accessibility value. Ambient occlusion therefore is an extreme simplification of spherical harmonics lighting, but is much easier to implement [PG04]. However, only rigid objects are covered. This may lead to artifacts in case grass billboards are deformed. Bunnell [Bun05] treats the polygon mesh as a set of surface elements in order to apply dynamic ambient occlusion. In each frame the rendering equation is performed over these elements, without testing for occluded directions. Instead of this, a shadow approximation function is used to solve the elements accessibility. A certain number of iteration passes over all surface elements are necessary in order to stabilize the results. An additional rendering of indirect lighting in real-time is possible, but the approach is too time-consuming to apply it to all grass billboards. Instead of just computing an ambient occlusion map, Cadet and Lécussan [CL07] precompute a static ambient occlusion Volume for the whole scene. The visibility information for dynamic objects in the scene is then interpolated across the volume's sample points near the object. A similar volume based approach for approximating the irradiance is made by Oat [Oat06].

2.1.3.2 Reflectance Model

In order to simulate reflection properties of grass applying the BRDF (Bidirectional Reflectance Distribution Function) model [MH99] is unsuitable. Boulanger et. al. [BPB06] used an approximation to BRDFs, so called BTFs (Bidirectional Texture Function), for their volume based approach to sim-

ulate the varying conditions between view direction and incident direction. Rather than using one semi-transparent image of the grass the texture includes more images each evaluated with another constellation of view to the incident light direction. Green [Gre04] used an approach to overcome the subsurface scattering problem of a light map shaded skin. Instead of using only one light map, an additional diffused version is added. Applying both the scattered and the light map itself, an illusion of scattering is produced. Kharlamov et. al. [KCS07] used a simple two-sided lighting model to illuminate leaves that are lit from behind. When the view direction and the light direction are opposing a linear interpolation between the transmitted color and the material color produces the final shaded color with respect to the angle between both directions. This model comes with no extra textures and it works for grass as well as for leaves.

2.1.3.3 Blending

Without visual blending between more or less transparent grass billboards the natural appearance especially of the grass edges is unpleasant. Alpha blending is a common algorithm to blend between semi-transparent objects, but due to the required depth sorting it is far from ideal. Instead, a more elaborated algorithm, the so-called screen-door-transparency can be used to avoid sorting [Wha05]. The alpha channel of a semi-transparent grass texture is modulated with a noise texture. Then the alpha test eliminates pixels from rendering and the human eye fills in the gaps between discrete samples. The Alpha-to-Coverage feature of modern graphics cards can be used to implement a similar effect. The resulting alpha value is used in a multisample resolution of the render target to decide how many subpixels will be written. Afterwards the blending occurs between the subpixels during the downsampling to the final resolution [Mye06].

2.2 Collision Detection

As the collision detection based on grass is less explored, collision detection algorithms on a wider range are examined in order to simulate interactive reaction based on objects moving through the scene. On a global scope many more or less specialized techniques come up with the problems of interference and collision detection. Several surveys to collision detection exist [LG98, Eri04, TKZ+04]. The collision detection usually consists of two phases due to the complex nature of the colliding set: The so-called '*broad phase*' to exclude non colliding objects on a coarser but also much faster scale and a so-called '*narrow phase*' where pairs of objects are checked for collision. Most of the latter techniques are using bounding volume hierarchies. Bounding volumes have been proven to be very efficient in the case of rigid objects. Several of them have been explored, the most appropriate ones are spheres [Hub96], axis aligned bounding boxes (AABB) [Ber97, LAM01], object oriented bounding boxes [GLM96] and discrete orientation polytopes [KHM+98, Zac98]. In case of deformable objects they have to be updated every frame. However, there is a great number of grass billboards which are stored and processed completely on the graphics memory and these cannot be

transferred to the main memory each frame to be tested against such volumes. For that case GPU accelerated techniques do solve the collision tests faster with bounding volumes. These are mainly based either on render targets or occlusion culling. Consequently, their accuracy is limited to the image resolution.

Sathe [Sat06] uses cube maps to approximate the shape of an object. In a preprocessing step the cube map is filled with distance values. These are distances from the center of the mesh to the outer shape in each direction. The vertices of one object are tested against the cube map distances of another object and vice versa during run time. On the one hand, all the tests are preformed on the GPU; on the other hand texture memory is heavily used. Both, the vertex buffers and the cube maps, must be available on graphics memory to be loaded in the shader process. Additionally, Sathe's approach is only useful for rigid objects.

In [KP03] the stencil buffer is used to test intersections. In the style of the shadow volume approach, at first, the penetrated object is rendered by writing the depth buffer. Then the penetrating object is rendered twice, the first time with active front faces and incrementing the stencil buffer and the second time the back faces are rendered while decrementing the stencil buffer. Subsequently, the rendered stencil values are tested: if the stencil value is not zero, an interference has occurred. The main drawback of this method is that each stencil buffer value needs to be checked and therefore a GPU memory read-back is necessary every frame.

A further approach is given by Heidelberger et. al. [HTG03, HTG04]. They perform the collision test in three stages. In the first stage the axis aligned bounding boxes of the intersection of two or more objects are computed. Furthermore, if an AABB exists the Layered Depth Images (LDI) for such a box are evaluated. This is done in an iterative process which is determined by the depth complexity of the object. Hence, a LDI is an array of depth-textures that represents the volume approximately. A LDI consists of a number of sorted depth values where each one belongs to a fragment of the object projected onto the texel. In a last step, the LDIs then can be used to determine if a vertex penetrates an object or if two objects collide. However, they require some buffer read-backs: The first copy is made to obtain the depth complexity and after generating the LDI there is another buffer read-back to sort the depth values.

Govindaraju et. al. evaluate a top down approach. Initially, they compute a potentially colliding set of objects with the aid of the graphics hardware. At the beginning all objects belong to the colliding set and then they are sequentially pruned away. The exclusion of one object is based upon hardware accelerated occlusion queries against the rest of the current colliding set. If an object is fully visible to one of the view directions along the world-space axes, it is not colliding and so will be pruned away. Finally, an exact triangle to triangle intersection test is performed on the CPU for the remaining objects to check whether collision occurs to them or not [GRLM03, GLM05]. However, the final collision test in this approach is realized on the CPU which leads to performance losses.

Kolb et. al. [KLRS04] and Vassilev et. al. [VSC01] also offered an approach to collision detection using depth maps which are fully generated and accessed on the GPU. The number of depth maps are

representing the outer shape of an object. At least each depth map stores distance values and normals. The collision test then is realized in the shader: At first the vertex position is transformed to the projection space of the depth map. After that a lookup into all depth maps occurs. Afterwards the depth of the transformed vertex is tested against the depth map distances to determine their position, inside or outside of the object. Similar to Govindaraju et. al., the vertex is assumed to be outside if at least one test is positive. The distance map approach seems to fit best for the very reason that all computations, including the reaction, are done on the GPU.

2.3 Cloth Simulation

Techniques are necessary for simulating deformable objects in order to overcome the problems of elongation on the trot of external forces which are applied to the grass billboards. Especially the cloth models are of interest. Therefore, Hauth et. al. [HEE+02] give a good overview of the physical model that underlies most cloth-based simulations. In addition, they weigh up the pros and cons of several methods of numerical integration. They also offer a method to render cloth with complex materials. The cloth model mostly consists of a spatial coined network of point masses. Each pair of adjacent masses is linked by different stiff springs. These springs are elongated or clinched with regard to the existence of external forces like collision or wind. In dependence on the stiffness of the springs a more or less strong reaction force tries to bring them back to an equilibrium. All external and internal forces are integrated over the time. In case of large time steps the stability of the system is determined by the integration method applied.

Baraff and Witkin published a cloth simulation model based upon an implicit numerical method to overcome the stability problems. A scalar energy function is used to accumulate the forces. Then the implicit Euler integration generates a linear system which is solved using the modified conjugate gradients method [BW98]. Even if GPU accelerated methods exist to solve these equations [BFGS03], the additional computational burden is unnecessary since other methods for interactive real-time applications are more practical.

Xavier Provot explicitly integrates the external and internal forces over time with the aid of the forward Euler method. He noticed a less realistic result in small regions of the cloth due to less stiff springs. A post processing step is made to correct their length [Pro95] in order to avoid an increase of the stiffness of springs which would result in more costly iterations. Fuhrmann et. al. replace the cloth forces by several length constraints along the connection of two particles in order to overcome the problem of large time steps. Hence, only the post correction steps, introduced by Provot, are needed. Then a few iterations over all constraints are performed, as the alteration of one of the springs affects neighboring springs as well [FGL03]. Zellner simulates a similar approach to Fuhrmann et. al.. He uses the stream output stage to recurse over the springs. As a result the constrained based cloth simulation is entirely offloaded to the GPU [Zel07].

Chapter 3

Modern Graphics Hardware

Graphic Processing Units (GPUs) are highly efficient parallel data processors. They have major advantages compared to current Central Processing Units (CPUs) whenever massive data can be parallelized and flow control mechanisms are less frequently used. In addition, after many transitions over the recent years, the Single Instruction Multiple Data processors of the graphics cards are now offering many programming capabilities of current CPUs and the rendering pipeline architecture has been evolved as well.

The techniques which are applied for the solving of responsive grass use the potentialities of the fourth generation of graphics cards. A short summary of the innovations with respect to the previous generations is presented in this chapter.

3.1 Graphics Pipeline

The upper part of Figure 3.1 shows the rendering pipeline which is arranged in several subsequent stages, with specific input and output restrictions. Following the prior pipeline (the yellow parts in Figure 3.1) the input assembler gathers vertex data form several streams and then the programmable vertex shaders projects them to the so-called clip space. The rasterizer builds up fragments with regard to the declared primitive type and the projected vertices. Afterwards these fragments are processed in the programmable fragment shaders. Finally the output merger writes the resulting pixel values to their frame buffer location, after passing several so-called fragment operations[1].

Although the previous version of the rendering pipeline can still be used the newly introduced programmable geometry shader stage (see Section 3.1.1) and the stream output stage (see Figure 3.1) both offer possibilities that are rather important especially for the implementation of the collision system and the rendering system of the grass layer.

[1] For a more detailed description of the previous graphics pipeline, have a look at [MH99, WNDS99, Gra03]

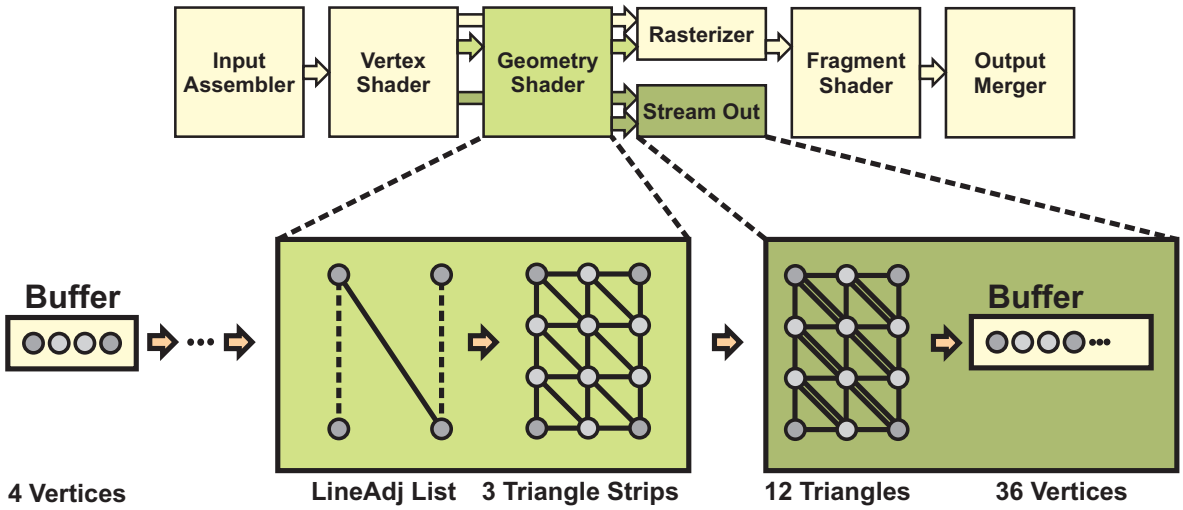


Figure 3.1: The rendering pipeline. The yellow path shows the prior rendering pipeline and the green parts are the newly added extensions. The geometry shader has the possibility to generate the final topology. In addition, the number of vertices might be amplified. The primitives are directly written to the graphics memory whenever the stream output stage is used.

3.1.1 Primitive Based Programming

The programmable geometry shader stage (see Figure 3.1) offers new programming possibilities based on primitives. The whole primitive is passed to the shader as an input [Dog07] (see the line adjacent primitive (LineAdj) in Figure 3.1). Moreover, the input assembler is expanded by new primitive types in order to hand over adjacent information to the geometry shader for each primitive [Bly06, BL06]. The geometry shader has the ability to operate on its vertices and finally amplifying the number of primitives by emitting more than one of them at each invocation [BL06]. This enables a handling of six vertices (in the case of a triangle adjacency list) in one shader invocation. Within certain limits, the geometry shader offers the possibility to create a multiple of the vertices [Bly06]. In addition, it is possible to write out a primitive type differing from the one passed on the input at the same time. This enables a creation of the final topology at this stage of the pipeline (note the triangle strips in Figure 3.1). This opens the possibility to refine mesh topologies during the rendering pipeline, for instance.

Furthermore the geometry shader is able to distribute the primitives to eight render targets simultaneously when using the rasterizer back-end. This allows the projection of each primitive to eight projection spaces in one single render call.

3.1.2 Streaming Architecture

Since the prior data flow does start with 1D vertex buffers and ends up by writing to 2D textures, a conversion is necessary due to the fact that the output and input formats are different from each

other. Consequently the update of vertices or 1D data on the graphics memory without the expensive read-back to the CPU has often been achieved by several renderable 2D textures [Sch06] because the internal processing was strictly bound to fragments. Furthermore, only a small number of frame buffers can be used as render targets at once which limits the size of each datum to one render call.

The stream output stage can be used in order to overcome these limitations for one-dimensional data as for example vertex buffers: The vertices are written to a 1D vertex buffer which resides on the graphics memory. This can be done directly after they are processed by either the vertex shader or the geometry shader without even using the rasterizer back-end [NX006]. Neither clipping, projection, primitive setup and rasterization nor the pixel operations take place. This shortens the updating process and allows an efficient update of the vertex data which requires only a minimum of CPU handling. The stream output stage supports much richer output formats than the output merger and additionally, the stream output buffer is much more flexible and larger than frame buffers. However, the streamed data is restricted to a size of sixteen tuples of one or four data items, for example float4 which means $16 \times 4 \times 4$ bytes [Bly06]. Furthermore, the so-called transform feedback [WC08] mode records the streamed data which can be queried by the CPU or can be used directly to process the streamed data in the next GPU pass without any extra CPU intervention. However, a buffer cannot be bound to both the input assembler and the stream output stage at the same time.

The output merger, a common technique for the blending of semi-transparent objects based on the fragment operations. Therefore an algorithm on the CPU has to sort all semi-transparent objects in the scene before rendering them. In contrast to this alpha-to-coverage solves the problem without expensive depth sorting [NX006]. Furthermore depth sorting algorithms can be avoided if no absolutely correct blending between semi-transparent objects is necessary. The alpha value is used to determine the number of subpixels that will be filled with the current pixel color. The blending between the subpixels is performed while resolving the multisample resolution to the final image resolution [Mye06]. Even if alpha-to-coverage is a feature provided by the API it uses the multisampling capabilities of todays graphics hardware.

3.2 Unified Shader Model

Prior programmable pipeline stages were built with a fixed number of stream processors which are designed to operate either on vertices or pixels. Thus, there was a fixed amount of vertex pipelines and a relatively large but although fixed number of pixel pipelines due to the fact that pixels are more frequent than vertices. If the stages are fixed they can only attain as much performance as shader units are available for the pipeline stage [NV006] as illustrated in Figure 3.2(a).

The GPUs dispatch logic of the least graphic card generation can assign vertex, geometry or pixel tasks dynamically to the available general purpose streaming processors [Dog07]. That is possible because all streaming processors have the same instruction set [Bly06] [NX006]. As a consequence, the implemented unified shader model is useful in cases where a heavy work load is assigned to

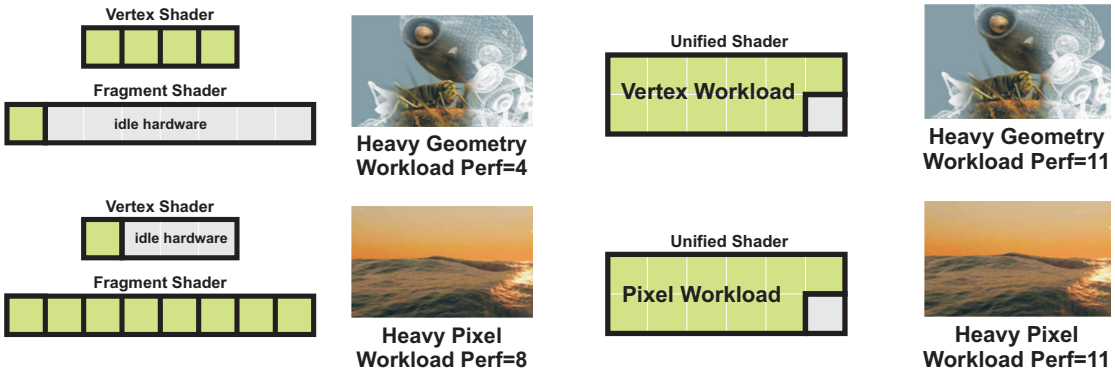


Figure 3.2: The advantage of a unified shader model. As can be seen in Figure 3.2(a), heavy workload on one streaming processor can not be offloaded to the other type of processors. Instead when a unified instruction set is offered the workload can be distributed over all available streaming processors as is illustrated in Figure 3.2(b).

one certain programmable stage. The other programmable stages are used less frequently (see Figure 3.2(b)), for example while streaming a large number of vertices by using the stream output stage as the back-end. In this case the fragment shader is not needed at all. Subsequently, the distribution of the computation is dynamic [NV006] and thus, the balancing of the shader pipeline is displaced to the dispatch unit.

In addition, the shader model supports texture arrays which yield more flexibility to the addressing of the texture memory: Textures stored in a linear arranged array are dynamically indexable in the shader [Bro08]. Rather important is the ability to bind each texture of a texture array as a render target in the output merger. However, tri-linear interpolation is not supported for them and in addition there is the restriction that at up to 1024 textures can be stored which have to be of the same resolution [NX006].

Chapter 4

The Animated Grass Layer

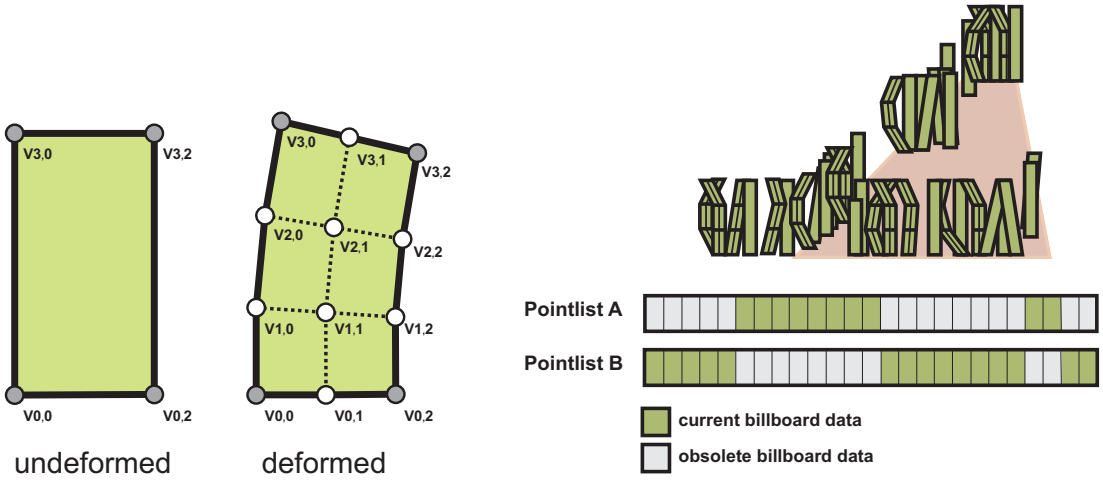
This chapter focuses on structural aspects of how the waving grass over arbitrary terrains is realized on the GPU. Therefore, four mayor topics have to be discussed: At first in Section 4.1, the grass billboards which are the base element of the responsive grass layer are introduced. In the next step, the spatial structure which divides the grass layer into more manageable tiles of grass billboards is presented in Section 4.2. As it is not handy to model each clump of grass separately, a generation process is applied which procedurally generates grass billboards in respect to the terrain's shape and some user-definable parameters. All this will be described in detail in Section 4.3. Finally, Section 4.4 describes the animation process which treats a basic property of grass or meadows, namely the response to wind.

4.1 Grass Billboards

Since a large area of the terrain is covered by grass objects and since they appear quite frequently, it is not convenient to model each blade of grass separately. Hence, clumps of grass are represented by semi-transparent decal textures which are projected onto quadrilateral objects similar to [Pel04], which results in the final look of the grass objects. Streaming respectively rendering each grass billboard in a separate render call overwhelms the CPU. That is why the grass billboards are stored across two large point lists as the GPU works best on data that can be processed in parallel.

4.1.1 Grass Objects

Figure 4.1(a) illustrates that a grass object can have two mesh representations in order to account for deformations which are caused by colliding scene objects. If no deformation has occurred, only a single quad forms the grass object as it can be seen on the left side of Figure 4.1(a). This quad consists of four edge vertices $\mathbf{v}_{0,0}, \mathbf{v}_{3,0}, \mathbf{v}_{0,2}$, and $\mathbf{v}_{3,2}$ [Pel04]. Whenever a collision occurs, the deformed representation is necessary (for more information see Section 5). Therefore, the mesh is subdivided into a 3×4 grid of vertices $\mathbf{v}_{i,j} \in \mathbb{R}^3$ with $j \in \{0, 1, 2\}$ and $i \in \{0, ..., 3\}$, as shown on the right side of Figure 4.1(a). In addition to the edge vertices the mesh includes inner vertices. It is



(a) (b)

Figure 4.1: The grass billboards. Figure 4.1(a) shows both representations of a grass billboard. In Figure 4.1(b) their memory layout is shown. All billboards of the grass layer are stored across two buffers on the graphics memory. A billboard only exists in one point list at a time.

important that all the vertices $\mathbf{v}_{i,j}$ are defined in world coordinate system. Furthermore, the vertices $\mathbf{v}_{0,0}, \mathbf{v}_{0,1}$ and $\mathbf{v}_{0,2}$ are fixed to the terrain. Consequently, only the vertices with $i \in \{1, 2, 3\}$ are able to respond to any type of force.

4.1.2 The Memory Layout

In general a vertex shader works on a single vertex at a time, and thus its output is also just a single vertex. The goal is to retrieve a mesh which consists of the billboard's vertex positions. The geometry shader is used to create the final mesh of the grass object during the rendering. This means that the complete grass layer is accessed by the graphics pipeline as a large point list[1]. Each point-element contains the whole information of one single billboard which is defined by the vertices as well as some state information described throughout the following sections. The structure of such an element is shown in code sample 1.

As it is not allowed to bind a buffer to both the stream output stage and to the input assembler during the same render call (see Section 3.1.2) a second buffer of the same size is necessary. Both buffers are created on the graphics memory. If the collision system needs to update the billboard data, one buffer is bound to the input assembler and the other is bound to the stream output stage. As a consequence of the streaming process one billboard exists only in one of this two buffers at each point in time. The other buffer contains obsolete data at the location of the billboard (see Figure 4.1(b)).

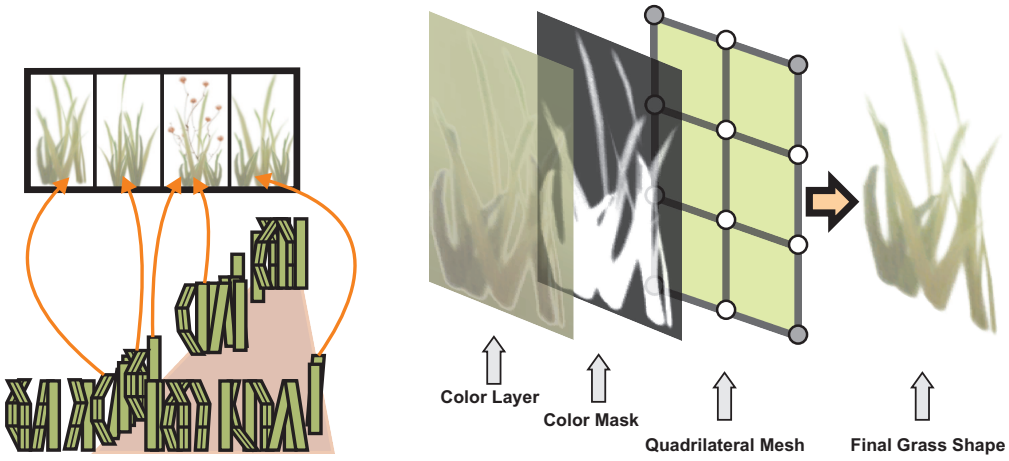
Moreover, the grass billboards are grouped into clusters which are called grass tiles due to the spatial octree layout described in short.

[1] Other primitives than points are able to manage the data for each billboard as well.

```
// Billboard Data Definitions
struct BILLBOARD_DATA
{
    float3 Vtx00            : VERTEX0;
    float3 Vtx01            : VERTEX1;
    float3 Vtx02            : VERTEX2;
    float3 Vtx10            : VERTEX3;
    float3 Vtx11            : VERTEX4;
    float3 Vtx12            : VERTEX5;
    float3 Vtx20            : VERTEX6;
    float3 Vtx21            : VERTEX7;
    float3 Vtx22            : VERTEX8;
    float3 Vtx30            : VERTEX9;
    float3 Vtx31            : VERTEX10;
    float3 Vtx32            : VERTEX11;
    float3 GrowDir          : GROWDIR;
    float3 SpringLens       : SPRINGLENGTH;
    float  RecTime          : RECTIME;
    float  ImageId          : IMAGEID;
};
```

Code Sample 1: The data layout. Vertex information $\mathbf{v}_{i,j}$, grow direction $\mathbf{d}_{\text{grow}}$, the initial spring lengths $\mathbf{s}$, recover time t_{rec} of the billboard and an index id_{image} addressing the decal texture are stored in a single element. Two point lists which are filled with these elements are stored on the graphics memory. By using this structure the maximum spread (sixteen tuples of float data) is occupied for an element which is bound to the stream output stage.

4.1.3 Grass Textures



(a) (b)

Figure 4.2: The grass textures. Each grass object has an index into the texture array as displayed in Figure 4.2(a). The semi-transparent decal images are randomly distributed over the grass layer. Figure 4.2(b) shows how the final shape of the clump of grass is obtained.

A semi-transparent decal texture is planar projected onto the billboard's quadrilateral mesh, in order to yield the final apperance of a clump of grass. The 2D texture contains a number of grass

blades. Therefore, the color layer provides the material properties of the grass whereas the alpha layer is used as a mask during the blending process (see Section 6.3). Transparent parts of the texture are used to cut off irrelevant areas of the color layer, as displayed in Figure 4.2(b). Several grass textures are randomly repeated over all grass billboards of the plant cover in order to achieve a sort of randomness. Thus, a texture array is applied (review Section 3.2) which provides an RGBA texture at each level. Additionally, each of the grass objects stores an index id_{image} into this texture array which is used to address the final decal texture at run time (see Figure 4.2(a))

4.2 Grass Tiles

Due to the fact that some billboards are not affected by collisions or that they might be invisible, the rendering of a entire array of the billboards turns out to be not efficient. Therefore, the grass billboards are organized in tiles which are constituted by a spatial octree structure. The bounding box of a tile is tested before the collision handling and rendering passes. This improves the performance enormously. However, the batch size of grass tiles has to be taken into account.

4.2.1 The Octree Structure

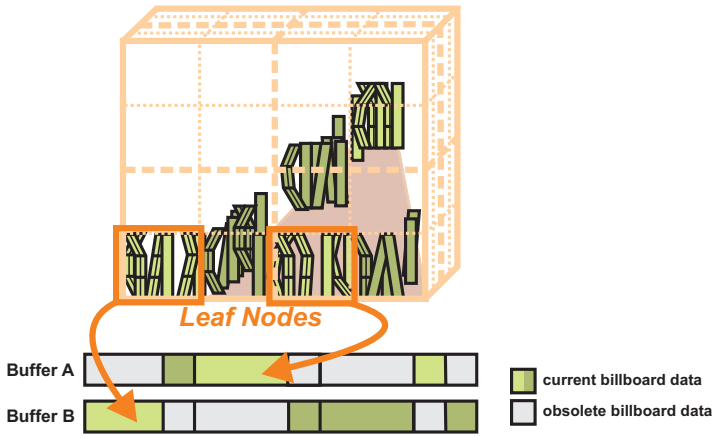


Figure 4.3: The octree structure. The grass billboards are assigned to the octree's leaf nodes.

It is important to process as many grass billboards as possible during each pass on the GPU for maximum efficiency. Thus, the grass layer is divided into disjunctive tiles of grass billboards by using an axis aligned grid which encloses the grass layer. In detail, the grid is defined by a hierarchical axis aligned octree structure [MH99]. Each *axis aligned bounding box* (AABB) of an octree level is subdivided recursively into $2 \times 2 \times 2$ subsequent child AABBs in order to build the hierarchy. It is important that each AABB of the tree, if it is not a leaf node, encloses its child AABBs. Each leaf node has an index range addressing those billboard's covered by the leaf node as shown in Figure 4.3. That is why the grass billboards are sorted by a pre-process with regard to the octree's structure.

All leaf nodes are stored in a linear memory structure to enable hash index computations [Eri04]. This octree hierarchy is necessary during the viewport culling of the render process (see Section 6.3). Additionally, the leaf node's size is restricted to be at least as large as the largest object handled by the collision system including the grass billboards. This prevents the collision system from missing a collision between the grass layer and the scene objects as described in Section 5.2.

4.2.2 Minimizing Render Calls

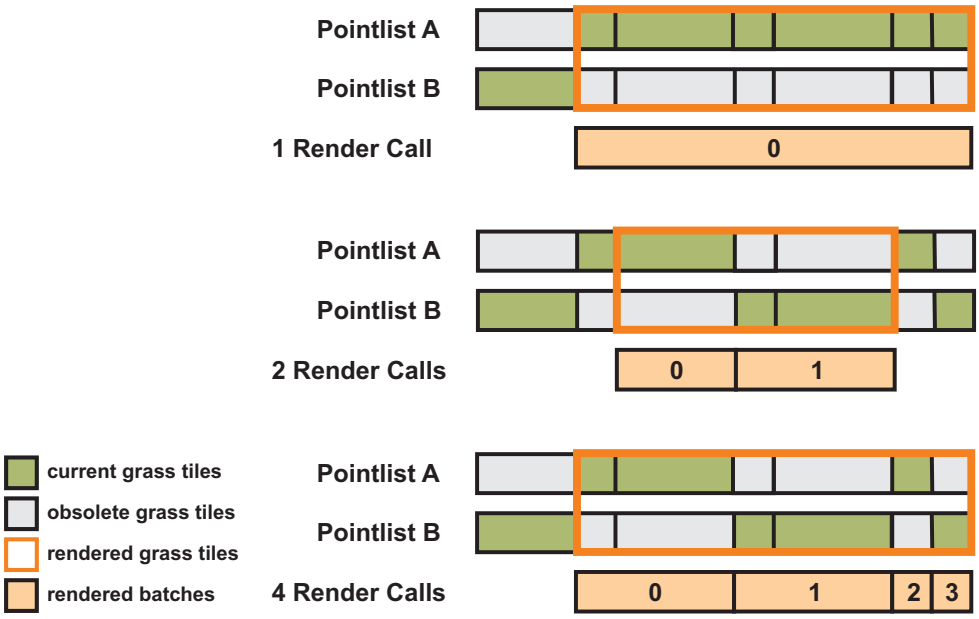


Figure 4.4: The batching process of grass tiles. Grass tiles that are adjacent in graphics memory can be combined to batches of greater size. As a result those batches are cause less system calls due to the GPU based handling as each of them can be passed to the GPU in a single render call. The data is spread over both buffers. In all cases there are less render calls if batches are passed to the GPU instead of rendering each tile separately.

During runtime the octree structure is used to decide which of the grass tiles should be processed. Furthermore, the billboard data is spread over both buffers as a result of the prior collision handling: Some billboards do exist in the swap buffer while others exist in the other one. However, it is not suitable to perform a single render call for each of the grass tiles. This may lead to a bottleneck caused by too many render calls.

Subsequently, ranges of billboards which are adjacent in graphics memory are organized to batches of grass tiles. These batches are passed to the GPU in a single driver call as shown in Figure 4.4. The more grass tiles can be grouped, the less system calls occur. The grass tiles which cannot be arranged to a coherent index range still have to be rendered in separate calls. This batch-process is applied whenever grass tiles should be handled by the GPU.

4.3 Procedural Generation

The grass layer often covers a large area of the terrain. Consequently, it contains thousands of grass billboards, each consisting of a simple geometry. It is thus obvious to apply a procedural technique on the GPU to generate the grass layer. Different *influence maps* are used to control the procedural technique to allow some user-defined design choices.

4.3.1 The Influence Maps

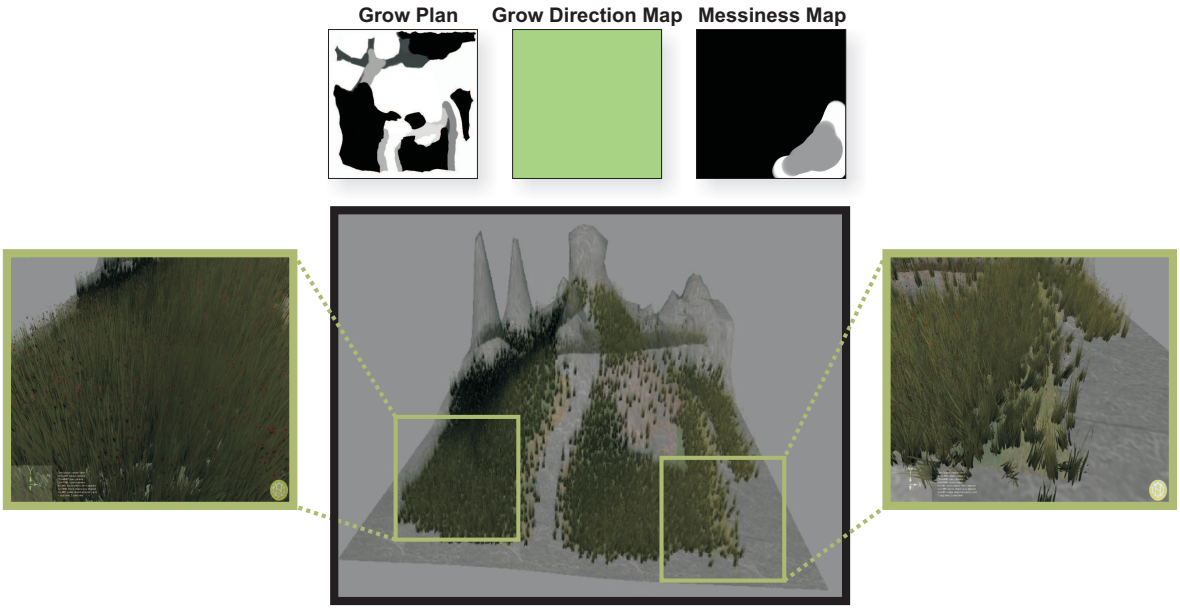


Figure 4.5: The influence maps. The texture images for the grow-plan, the grow-direction and the deviation, and the resulting plant cover.

Even if the grass layer is procedurally generated, though, it is necessary to control some local visual properties while maintaining the generation as user-friendly as possible. Therefore, a set of 2D maps is used to manage some of the design goals. Unique texture coordinates are spread over the vertices of the terrain's mesh in order to provide a unique value for each of the terrain's primitives. These are used to sample the maps at the location of the mesh's vertices.

The map which is applied first is called grow plan and as diplayed in the upper left of Figure 4.5. The texture defines local scalar densities of the grass layer which work in a similar manner like the density map used by Boulanger et. al.[BPB06] but without the restriction to be applied at runtime. The higher the density, the more grass billboards are planted on the tile of the terrain. Moreover, the sampled values of the grow plan are used to fade out the amount of grass billboards in order to simulate a crossing between fertile and barren ground. A second non-scalar map called grow-direction map provides normalized 3D directions which define the orientation for each of the grass billboards. Furthermore, another scalar map called messiness-map, is used to provide an amount for

the randomness in terms of the grow direction. This map influences the amount of rank growth. Since the geometry shader creates the plant cover all the maps are coarsely sampled at the positions of the terrain's vertices. The resulting plant cover which is based on the maps is shown in Figure 4.5.

4.3.2 The Generation Pipeline

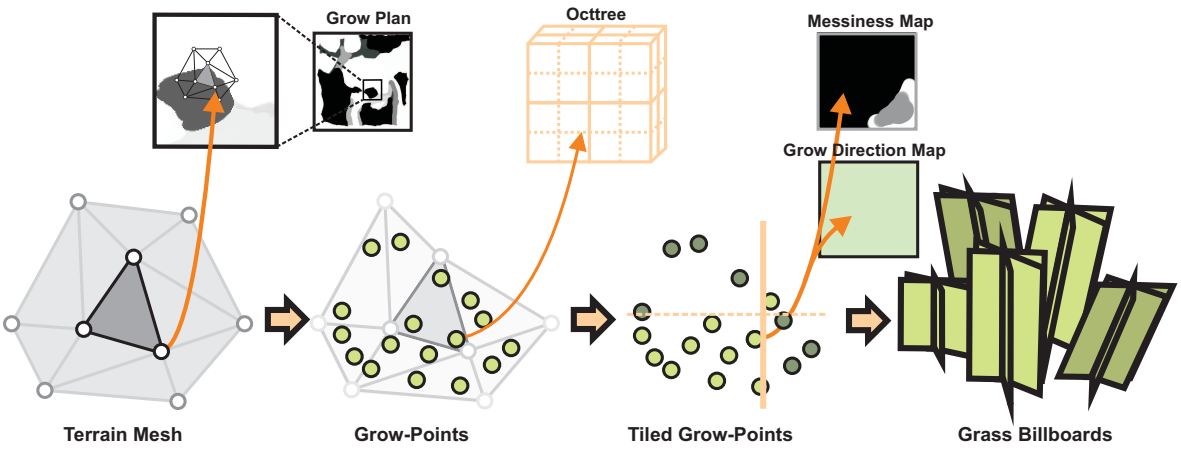


Figure 4.6: The generation pipeline. In the first pass the grow plan is used to generate base points on the terrain's mesh. Next, the octree structure is set up. Finally, the grass billboards are planted on each base point. Furthermore, the grow-direction map and the messiness-map are sampled in order to align the crossed billboards.

The generation pipeline is shown in Figure 4.6. As the number of billboards is not known when starting the procedural process the first step is the evaluation of the final number of grass billboards and their positions in respect to the user defined grow map. This set of base points is computed on the GPU and finally read-back to the CPU after each primitive of the terrain is processed. Based on this information the final hierarchical octree structure is build. The base points are distributed over the octree's leaf nodes. As a result each of these nodes contains a set of base points. Afterwards, a geometry shader creates a set of crossed billboards on each of the leaf node's base points. The procedurally generated grass billboards are then streamed to the graphics memory.

4.3.2.1 Plant Cover Information

In a first step of the generation the final number of grass billboards as well as their positions are obtained. Therefore, the grow plan is sampled for each triangle of the terrain's mesh. According to the sampled value a series of base points is randomly placed on each triangle by using barycentric interpolation between the triangle's edge vertices. Furthermore, a texture coordinate is interpolated for each base point. The base points are streamed to the graphics memory. In addition, the number of streamed points is recorded by the graphics hardware and can be obtained by a stream output query (see Section 3.1.2). Since the final number of billboards is known, two point lists which store the

billboard data (view Section 4.2.2) are allocated on the graphics memory.

The size of the terrain's primitives influences the density of the plant cover as the operation is applied on the basis of triangles. As a consequence the grow plan has an indirect impact on the density. This is because the same density value may result in a dense grass cover for a small primitive and a sparse cover for a large primitive. However, the generation of the plant cover information yields good results in case of planar areas where the size of the triangles is almost uniform.

4.3.2.2 Spatial Clustering

After the base points are read-back to the CPU the hierarchical octree structure is constructed. Each leaf node of the tree should at least be as large as the largest object handled by the collision system. Therefore, either the maximum size of the responsive grass billboards which is defined at startup or the maximum size of the dynamic collision objects determines the size of the leaf nodes. As a consequence, the size of the scene objects must be available at this point of the generation pipeline. Furthermore, the size of the octree is determined by the number of nodes which are necessary to cover the whole grass layer.

In the next step the base points are distributed over all the leaf nodes. Each leaf node receives those base points that are covered by their bounding box. As a result each leaf node has a list filled with the covered base points. Afterwards, each of those lists is again stored in a vertex buffer on the graphics memory in order to generate the final billboards on the GPU. Furthermore, an index into the billboard buffers is assigned to each leaf node. During runtime this index range makes it possible to address the grass billboards that are covered by a node.

4.3.2.3 Procedural Grass Billboards

The final generation of the grass billboards is executed on the GPU. Each grass tile is generated in a separate geometry shader pass. As a result a set of crossed grass billboards is built at each of the leaf node's base points. Figure 4.7 illustrates the required steps. Furthermore, the generated information of each billboard is streamed directly to one of the point lists which are used during runtime.

For each of the base points the normalized direction $\mathbf{d}_{\mathrm{grow}}$ in which the billboard should be extended is looked up into the grow-direction map. The messiness-map is also sampled in order to obtain the deviation angle to the grow-direction. The higher the sampled messiness value is, the more the grow-direction is rotated. Therefore, a randomized rotation axis $\mathbf{d}_{\mathrm{ortho}}$ orthogonal to the grow direction $\mathbf{d}_{\mathrm{grow}}$ is determined. Both the new grow-direction and the orthogonal rotation axis define the orientation of the billboard (see step three in Figure 4.7). The billboard's edge vertices are computed with regard to a randomly determined width w and height h (see step four in Figure 4.7). The inner vertices are necessary to define the deformed representation and they are not computed until a deformation becomes possible (see section5.2.3). Nevertheless, the initial extent of a deformed quad is stored in order to allow for shape preserving computations after a collision response (see Section 5.2.3). Therefore the width s_0, the height s_1 and the diagonal length s_2 of a quad of the

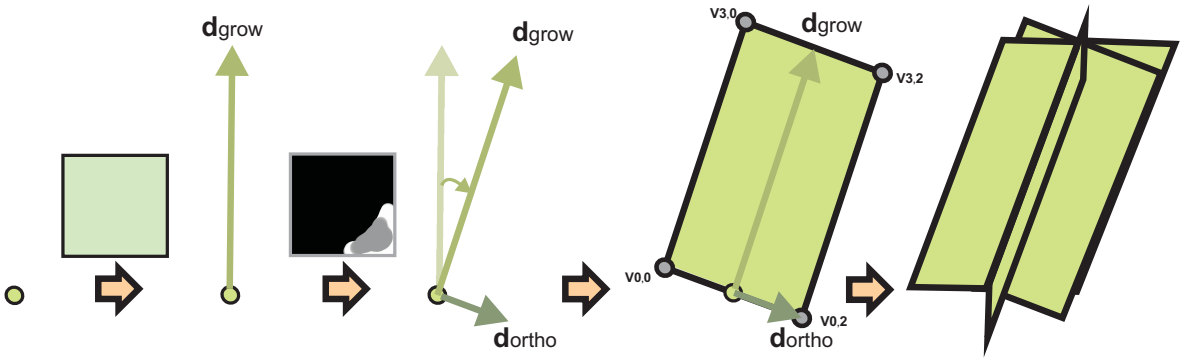


Figure 4.7: The procedural billboard generation. For each base point a grow-direction is looked up in the grow-direction map. The messiness value determines the deviation of the grow direction. Finally, the crossed billboards are generated along the grow direction.

billboards deformed mesh are stored within the billboard data. As the final shape of the clump of grass is achieved due to the projected semi-transparent decal texture, a random texture index id_{image} addressing the texture array is also assigned (Please note the billboard's data layout shown in code sample 1).

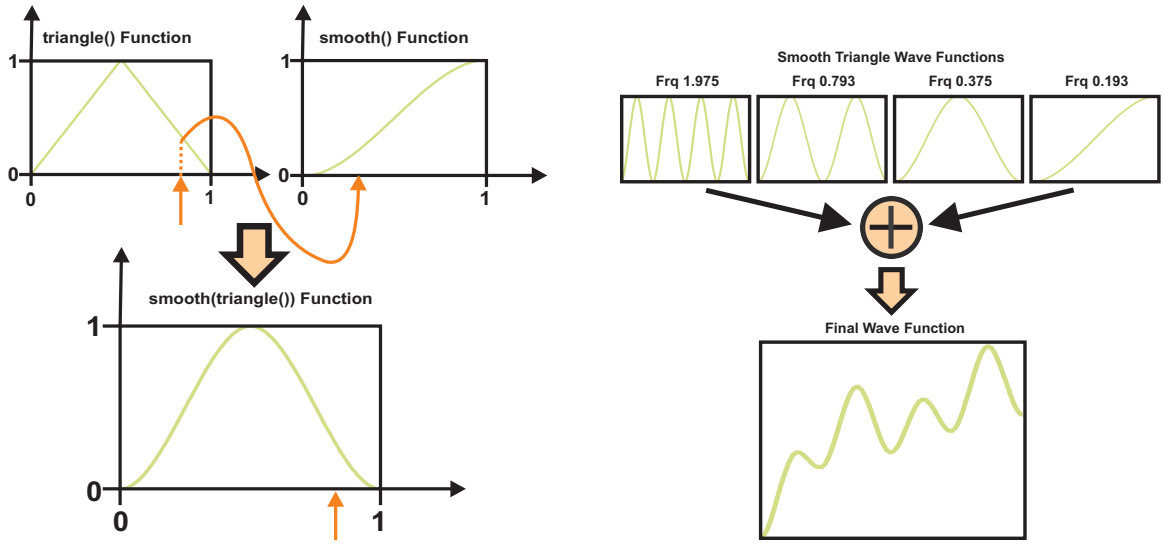
As the flat structure of the billboards is easy to estimate when viewing the billboards from their side, for each plant position three grass billboards are generated and crossed [Pel04] as illustrated in the last step of Figure 4.7. As a consequence the illusion of depth increases. However, for large regions where only a small number of grass billboards is planted, the flat structure is still estimated. After all three billboards are arranged their data is streamed through the point array stored on the graphics memory.

4.4 Wind Animation

As it is the movement of the grass blades in the wind what gives grass its natural and vivid look, a major key feature to all grass simulations is the way they react to wind forces. Keeping in mind that thousands of billboards have to be animated, the animation technique should not be too time consuming. Therefore, a sum of sinus approximations along the wind direction yields a translation vector which is applied to the upper vertices of each billboard. This results in a periodically movement along the wind direction which takes local differences over the grass layer into account. Furthermore, the wind animation is applied either during the collision handling or during the rendering process which are described throughout the following chapters.

4.4.1 The Wind Translation

For the periodic movement an approximation to the sine function is used in order to achieve a realistic animation. The so-called smooth triangle wave function [Sou07] is used to produce a translation vector which is applied to the upper vertices of the grass billboards.



(a) (b)

Figure 4.8: The periodic wind function. Figure 4.8(a) shows how the output of the triangle wave function is redirected as an input to the smooth function to yield the final value. Figure 4.8(b) displays the final wave function $\lambda(x)$ as the result of summarizing four of this concatenated functions.

The approximation of sine waves is achieved by concatenating two functions smooth$(x) \in [0,1] \in \mathbb{R}$ and triangle$(x) \in [0,1] \in \mathbb{R}$. The periodic property is satisfied by the triangle function as can be seen in the upper left graph of Figure 4.8(a):

$$triangle(x) = |frac(f \cdot x + 0.5) \cdot 2 - 1| \ , \tag{4.1}$$

with $x \in \mathbb{R}$ as the time-dependent parameter. The frac (x) function returns the fractional part of x and $f \in \mathbb{R}$ is the frequency. The range of the periodic repetition is $[z * f, (z+1) * f]$ with $z \in \mathbb{Z}$. Even if the function is periodic, though, the passage from high to low values and vice versa is not smooth at all. Therefore, the returned value is used to look up the final value in a cubic function as shown in Figure 4.8(a):

$$smooth(x) = 3x^2 - 2x^3 \ . \tag{4.2}$$

As a result the gradient of the triangle function is smoothed. So, the smooth periodic function is obtained by concatenating Equation 4.1 and Equation 4.2:

$$stw(x) = smooth(triwave(x)) \ .$$

Even if it is only an approximation to the sine function it is less time consuming[Sou07]. For a better understanding the function is plotted in Figure 4.8(a).

Summing up four of the smooth triangle wave functions, while providing four different static frequencies $f \in \{1.975, 0.793, 0.375, 0.193\}$ to each of them, yields the final periodic wave function provided by [Sou07] (see Figure 4.8(b)):

$$\lambda(x) = \sum_{k=0}^{3} stw_k(x) \ , \tag{4.3}$$

where x depends on several parameters:

$$x(t, \mathbf{w}, \mathbf{p}) = t \cdot s_{\text{wind}} + \mathbf{d}_{\text{wind}} \bullet \mathbf{p} \ . \tag{4.4}$$

As a first parameter the current time $t \in \mathbb{R}$ is passed. The second parameter is the wind force $\mathbf{w} \in \mathbb{R}^3$ to account for the wind direction $\mathbf{d}_{\text{wind}} = \frac{\mathbf{w}}{\|\mathbf{w}\|} \in \mathbb{R}^3$ and the wind strength $s_{\text{wind}} = \|\mathbf{w}\| \in \mathbb{R}$. So far, using only these parameters leads to an identical animation over the entire grass layer, since the parameters t, $\mathbf{d}_{\text{wind}}$ and s_{wind} are shared for all the grass billboards. The position $\mathbf{p} \in \mathbb{R}^3$ is added in order to incorporate local differences to the grass layer. The position differs for each billboard. There is still a less noticeable symmetry along the wind direction. However, this can be neglected as it is hard to identify.

Finally, the concatenation of Equation 4.3 and Equation 4.4 yields the translation strength which then in combination with the wind direction is used for the animation:

$$\text{wind}(t, \mathbf{p}) = \lambda(x(t, \mathbf{w}, \mathbf{p})) \cdot \mathbf{d}_{\text{wind}} \ . \tag{4.5}$$

4.4.2 The Billboard's Animation

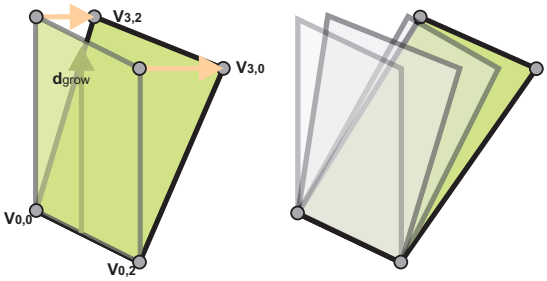


Figure 4.9: The billboard's wind animation. By using the grow direction the billboard's upper edge vertices are waving along the wind direction with respect to the translation strength. The directed translation strength $\text{wind}(t, \mathbf{v}_{0,j})$ is shown in orange.

The animation is reduced to a translation of the billboard's vertices along the wind direction. Only the undeformed billboards are involved in the animation process. If a deformation occurs the recovering process takes over the animation of the deformed billboards (see Section 5.2.3). Thus, if no deformation happens just the vertices $\mathbf{v}_{3,0}$ and $\mathbf{v}_{3,2}$ are subject to wind forces, as the vertices $\mathbf{v}_{i,j}$ where $i = 0$ are fixed to the ground.

```
inline
float4 Smooth( float4 x )
{
    return x * x * ( 3.0 - 2.0 * x );
}

inline
float4 Triangle( float4 x )
{
    return abs( frac( x + 0.5 ) * 2.0 - 1.0 );
}

inline
float3 Wind( in float3 p )
{
    // Compute the phase shift for the position p with respect to
    // the current wind strength and direction
    float phase = ( Time * WindStrength ) + dot( Wind, p );
    // Compute the four translation strengths.
    float4 ts = Smooth( Triangle( Frequencies * phase ) );
    // Compute the mean of the four values and
    // return the translation vector.
    return Wind * dot(ts,0.25);
}

inline
void ApplyWindForce( inout float3 Vtx[VTX_CNT], in float3 GrowDir )
{
    // move the upper vertices of the undeformed billboard
    Vtx[IDX_30] = Vtx[IDX_00] + GrowDir + Wind( Vtx[IDX_00] );
    Vtx[IDX_32] = Vtx[IDX_02] + GrowDir + Wind( Vtx[IDX_02] );
}
```

Code Sample 2: The wind animation. The wind animation moves the upper two edge vertices along the wind direction by summing up four translation strengths. The values **Wind**, **WindStrength**, **Time** and **Frequencies** are constant for each frame. **Frequencies** is a float4 which stores four different frequencies.

In order to translate the upper two vertices of the undeformed representation of the billboard, the initial grow direction $\mathbf{d}_{\text{grow}}$ of the billboard is required. The grow direction was stored for each billboard as an additional information due to the procedural generation process (see Section 4.3.2.3). The grow direction and the fixed ground vertices enable the restoring of the initial mesh of the billboard. The reconstructed mesh then is animated with regard to the current wind translation at the time $t \in \mathbb{R}$ (see Figure 4.9):

$$\mathbf{v}_{3,j} = \mathbf{v}_{0,j} + \mathbf{d}_{\text{grow}} + \text{wind}(t, \mathbf{v}_{0,j}) \,, \tag{4.6}$$

with $j \in \{0, 2\}$. As a result different translations for both of the upper vertices are applied. The code sample 2 shows the implementation of the wind animation. Note that the wind strength used for function 4.5 has to be choosen carefully. A translation which is too strong causes visual unpleasant distortions. Length preserving constraints are applicable in order to overcome the problem of distortions.

As a result the animation varies over the time with regard to the wind force and the fixed local ground positions of each billboard. Moreover the animation is independent to prior translations applied to the upper vertices of each billboard. As a consequence this enables a reconstruction of the billboard's shape after a deformation (see Section 5.2.3). It should be remarked that the wind animation is applied to each grass billboard separately without considering the crossed alignment.

Chapter 5

The Collision System

The collision handling of the billboards is the key feature to enhance immersiveness. The grass should yield a good reaction to dynamic collision objects. In addition, the process should also not be too time consuming. The billboards are solely processed on the GPU as the grass layer does not affect any scene object. Due to this fact the objects which cause the deformations have to be stored on the GPU as well. That is one of the reasons why implicit models are employed to represent the collider objects during the collision handling as described in Section 5.1.

The collision system is a cooperation between a CPU based ”broad phase” working on the spatial organized grass tiles and a GPU based ”narrow phase” working on the grass billboards whenever a grass tile is affected. The leaf nodes of the octree structure are used to reduce the collision handling based on the GPU side. In addition, a recovering process is controlled by both phases as well. The pipeline is describe in detail throughout Section 5.2.

The billboards are deformed by the collider object if a collision occurs. Subsequently, a recovering process brings deformed billboards back to their original shape. The billboard quads which are not affected by a collision can directly be rendered. In consequence, the overall performance of the animated grass layer is preserved dependent on the current recover time and the current collisions occurred. The implementation of the billboard’s collision handling is described in Section 5.2.3.

5.1 Implicit Collider Object Representations

The collision detection and its reaction based on the polygonal representations of complex objects causes computations which are far from ideal. As a consequence the collider objects are represented by implicit image based models as they are optimal to be handled on the GPU. Moreover, implicit representations have the advantage that distances are directly given which speeds up the penetration tests with the grass billboards.

Two different types of implicit objects are common: At first, bounding spheres are used to avoid unnecessary collision tests. After pre-decision, subtle tests are made based upon so-called depth cubes. These depth cubes consist of six maps which store relative distances as well as surface nor-

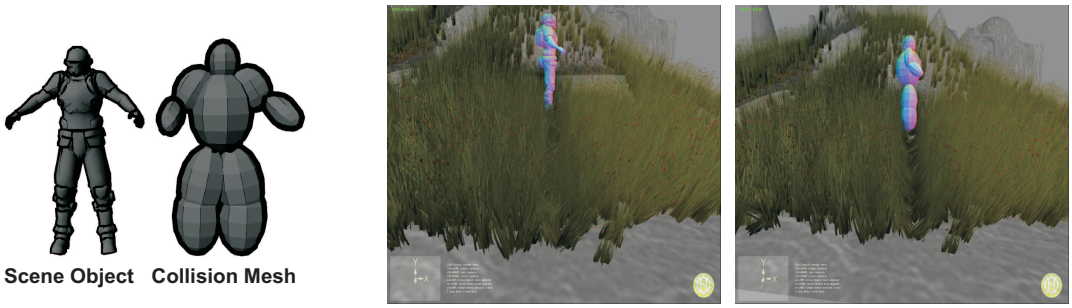
mals of the collider object. The depth cubes are generated by applying a more coarser polygonal representation which is called collision mesh in order to prevent the grass billboards from unnatural reactions.

5.1.1 Bounding Spheres

Bounding spheres completely enclose the object's geometry which they represent. Thus, they are used to speed up the collision handling. The efficiency is based upon their simple implicit formula which can be used to check for collisions between objects [MH99]. Throughout the collision pipeline they prevent further collision handling for grass billboards that are not penetrated by any bounding sphere (see Section 5.2.3). Furthermore, they are available to the GPU by using constant registers as described in Section 5.2.1.1.

5.1.2 Depth Cubes

As the bounding spheres are only usable for coarse preclusions, a more subtler implicit representation of the object is employed. The implicit representation is more efficient on graphics hardware due to the high parallel architecture of the GPU. Therefore, a coarser mesh called collision mesh is used to generate the implicit distances in order to overcome the problems which are caused by fine structures of the dynamic collision object. The collision mesh is projected to each of the faces of its bounding box. As a result a set of six maps is generated, each one providing relative distances from its near plane to the surface of the collision mesh. In addition they store the surface normals of the projected primitives. The information is required for a proper collision detection and collision response as described in Section 5.2.3. This set of textures is called the depth cube of the object [KLRS04].



(a) (b)

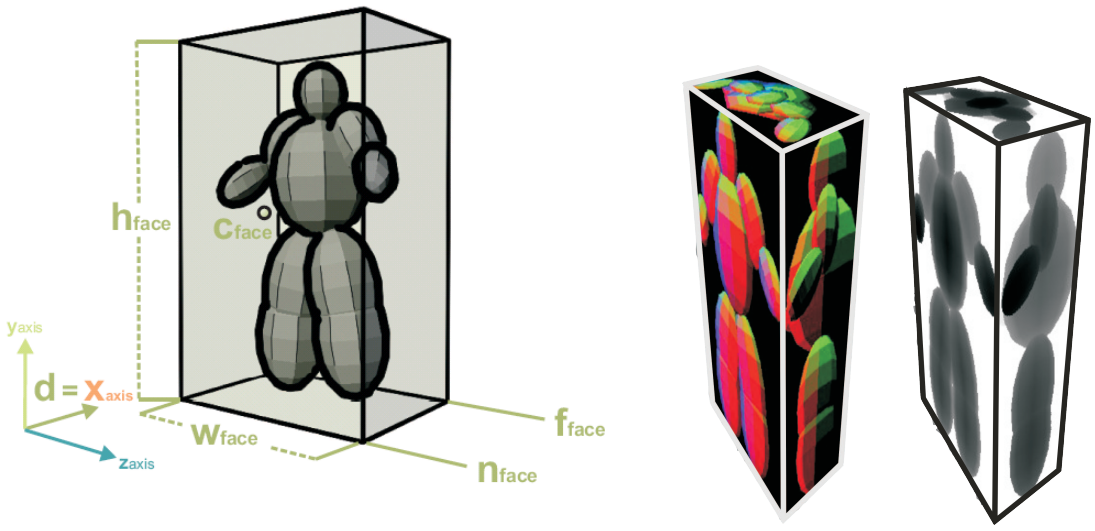
Figure 5.1: The collision mesh. Figure 5.1(a) shows the scene object and the corresponding collision mesh. On the left of Figure 5.1(b) the response with the mesh of the scene object is shown. The object is moved away from the viewer. On the right the response is shown if the scene object is replaced by its coarser collision mesh. Note the clearly perceptible reaction in contrast to the reaction which is caused by the mesh of the scene object.

5.1.2.1 The Collision Mesh

The deformed mesh of the billboard is still coarse in contrast to the fine structures of the collision objects, for example, thin extremities of characters. Possibly, the collision handling based upon the vertices of the billboard (see Section 5.2.3.4) computes an unnatural reaction. Therefore, the cube map is created upon a coarser polygonal mesh, called collision mesh, as shown in Figure 5.1(a). Although, the sphere-like mesh is much coarser the results of the collision response are more pleasant as illustrated in Figure 5.1(b).

The depth cube of dynamic objects is updated in each frame. In case that an animation is applied to the scene object it is also necessary to animate the collision mesh. However, with the restriction of a much higher memory usage these maps can be precomputed if the animation cycles are known [VSC01].

5.1.2.2 The Distance Maps



(a) (b) (c)

Figure 5.2: The depth cube. The parameters to generate a distance map are shown in Figure 5.2(a) for the projection direction $\mathbf{d}_{proj} = (1, 0, 0)$. Figure 5.2(b) shows the resulting normal information mapped to the RGB color range. The resulting distances are shown in Figure 5.2(c). Darker areas are closer to the projection planes than bright areas

The six distance maps of the depth cube are generated on the GPU by simply placing an orthogonal camera to the center of each face of the object's axis aligned bounding box. In detail, the distance maps are the result of the projection of the mesh onto each face. The parameters which are necessary to define the projection are shown in Figure 5.2(a). For each of the six 2D distance maps $DM_m,\ m = 0, ..., 5$ the projection plane is set to the near face n_{face}. The near face is determined with regard

to the distance map's projection directions. In addition, the far clipping plane is set to the far face f_{face}. Each projection direction $\mathbf{d}$ is one of the six normalized directions along the world space axis. Furthermore, width and height of the projection plane are set to the width w_{face} and height h_{face} of the current bounding box face. The origin of the projection space $\mathbf{c}_{\text{face}}$ is located at the center of the near plane. These parameters yield an orthographic projection transformation $\mathbf{T}_{WC\rightarrow DM} \in \mathbb{R}^{4\times4}$ (the transformation is described in detail in appendix A.1) which is applied to the vertices of the collision mesh:

$$\mathbf{v}^{'} = \mathbf{v}\ \mathbf{T}_{WC\rightarrow DM}\ .$$

As a result of the projection, the vertices $\mathbf{v} = (v_x, v_y, v_z, 1) \in \mathbb{R}^4$ are transformed from the world space to the projection space of the distance map. Thereafter, the $v_z^{'} \in [0, 1]$ coordinate is the relative distance of the vertex $\mathbf{v}^{'} = (v_x^{'}, v_y^{'}, v_z^{'}, 1)$ to the face of the bounding box. The value is written to the distance map in respect to the coordinates $(v_x^{'}, v_y^{'}) \in [-1, 1]^2$ (The projection of a single distance map on the GPU is described in detail in appendix A.2). The distance maps of the depth cube are shown in Figure 5.2(c).

In addition to the relative distances which are important for an appropriate collision test, the surface normals are stored for each distance map (see Figure 5.2(b)). These surface normals are used to respond to collisions [KLRS04] (see Section 5.2.3). Defining the triangle primitives of the collision mesh by their edge vertices $\mathbf{v}_0$, $\mathbf{v}_1$ and $\mathbf{v}_2$ the normal $\mathbf{n}$ for each primitive is computed by a cross product between the triangle's edges:

$$\mathbf{n} = (\mathbf{v}_2 - \mathbf{v}_0) \times (\mathbf{v}_1 - \mathbf{v}_0)\ ,$$

where $\mathbf{v}_i\ , i = 0, 1, 2$ are the world coordinates of the primitive's edge vertices. In addition, they are normalized in order to avoid different lengths which may lead to unpredictable reactions. Accordingly, a lookup at pixel coordinate (x, y) into a distance map of the depth cube returns a surface normal with unit length $\hat{\mathbf{n}} = \mathbf{n}/\left\|\mathbf{n}\right\| \in [-1, 1]^3$.

The distance maps are updated every frame in order to account for possible animations or rotations of the collider objects. As the geometry shader can distribute primitives to eight render targets, one single render call is sufficient to update the depth cube. The depth cube is stored using a texture array which is bound as a render target during projection.

5.2 The Collision Pipeline

The collision pipeline is split into two phases which are working on different levels of the grass layer as can be seen in Figure 5.3. Several processes divide the collision handling of the grass billboards into subsequent passes. These avoid unnecessary computations on the basis of each billboard. On the top level the collision handling on the CPU is based on each leaf node of the octree structure (see the

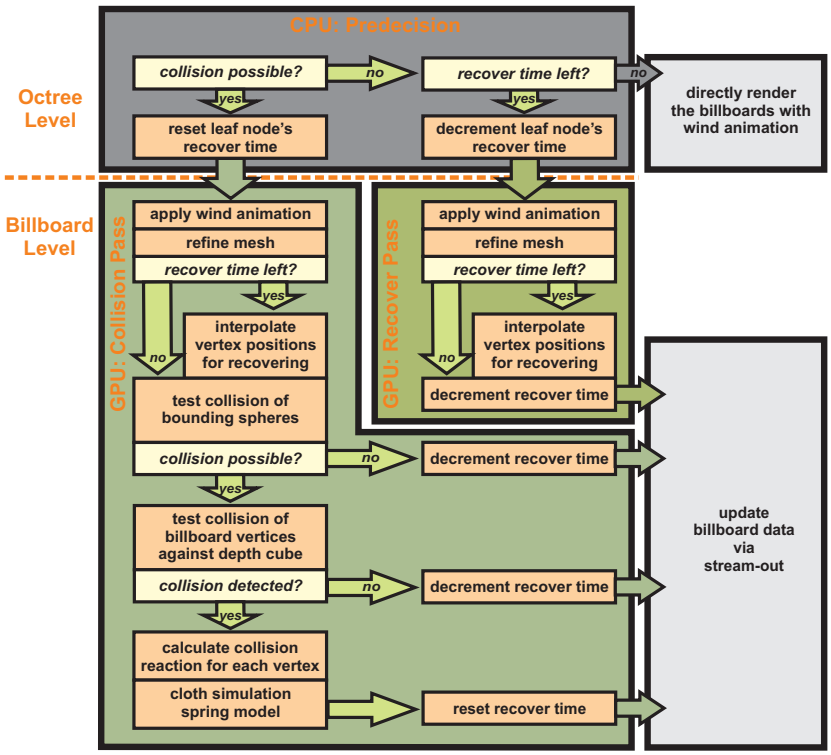


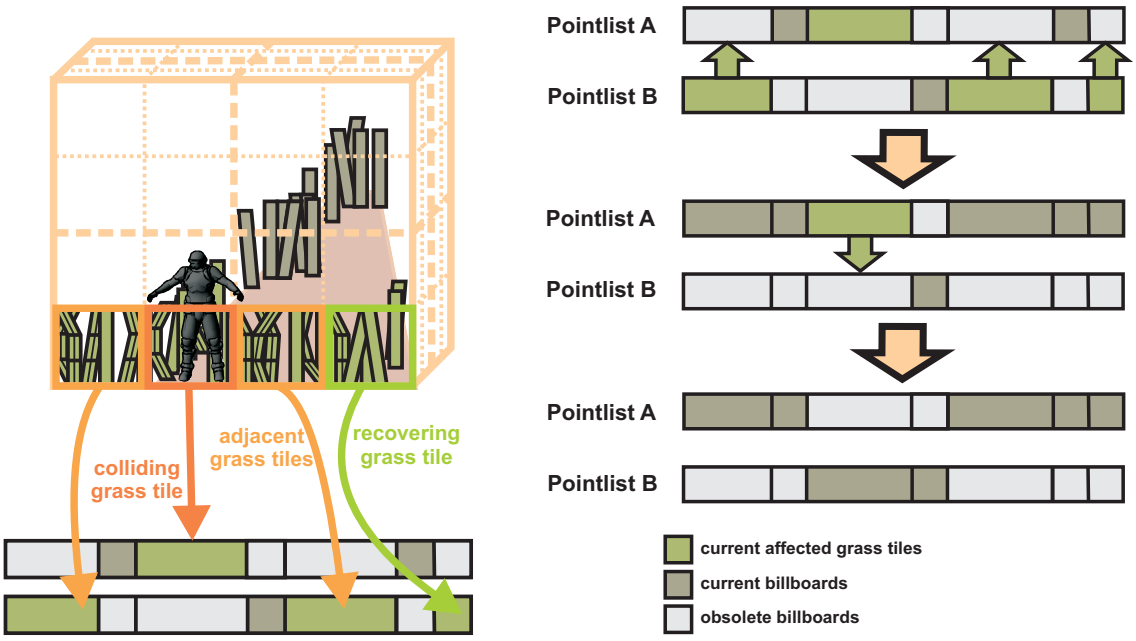
Figure 5.3: The collision pipeline. The collision handling is splitted into a CPU based part working on grass tiles respectively octree's leaf nodes and two GPU based parts handle collisions for the grass billboards. The decisions made on the CPU affect the rendering of the billboards.

dark grey box in Figure 5.3).

The GPU handles collisions on the basis of each billboard. In detail this depends on the decisions which are made by the CPU: The grass tiles which collide are updated by the collision pass highlighted by the left green box in Figure 5.3. Those tiles which still have to recover are updated by a separate recover pass. The recover pass is highlighted by the dark green box on the right in Figure 5.3. Both passes are covering the billboard's collision handling as described in Section 5.2.3. The grass billboards of the affected tiles are updated via the stream output stage. Those which are not updated in any of the two GPU passes are directly rendered. In that case the collision system does not have any effect on them.

5.2.1 CPU-Based Predecision

At the beginning of the collision pipeline, a coarse pre-test for the collision is preformed on the CPU. This test is highlighted by the dark grey box on the top in Figure 5.3. The grass layer is tiled by the octree structure and thus, each leaf node is tested to be affected by a collider object (see Section 4.2). According to the results of this test and the current remaining recover time, a node is marked either as colliding, non-colliding or recovering. For the colliding nodes the recover time is reset. Furthermore, a detailed collision detection, reaction and recovering on the basis of the grass billboards is performed



(a) (b)

Figure 5.4: The CPU-based handling. Figure 5.4(a) shows the colliding (red) and recovering (green) leaf nodes. The data of the corresponding grass billboards is updated via the stream output stage as shown in Figure 5.4(b). Both buffers swap their affected grass billboards in two subsequent steps. During the data streaming the billboards are updated via the geometry shader.

during the GPU based collision pass. Those non colliding leaf nodes which have some recover time left are marked as recovering. As a consequence, some billboards of the leaf nodes might be deformed and so still need to recover. These tiles are updated by the GPU based recover pass. If a leaf node is not subject to any further collision handling, it can be directly rendered.

5.2.1.1 Colliding Grass Tiles

The spatial octree structure of the grass layer is used to avoid collision tests based on the GPU. The centroids of the collider objects are used to determine those grass tiles which might be influenced.

Due to the arrangement of the octree's leaf nodes the search for affected nodes is reduced to a single lookup in the octree. A hash index of the leaf node is computed with respect to the position of the object's centroids. The hashed leaf node is marked as colliding. Additionally, their adjacent nodes are marked as colliding. These nodes are shown in Figure 5.4(a). Finally, a minimal set of colliding grass billboards is found. This is explained by the fact that all the leaf nodes are created in a size that leads back to the maximum size of either the collision object's bounding spheres or the maximum extent of the grass billboards (see Section 4.3.2.2). All gathered grass tiles are streamed throughout the collision pass by using as few as possible render calls (review Section 4.2.1). Before

they are updated, their recover time is reset.

During the collision pass the current scene object representations are required (see Section 5.1). For each of the batches of colliding grass tiles the corresponding depth cubes and bounding spheres are collected. Each depth cube as well as bounding sphere is passed to the GPU before the batch of grass tiles is streamed. In addition, for each collider object the size of the axis aligned bounding box and each of the six projection transformations are passed. The bounding box size is used to map the relative distances of the depth cubes to world space distances which are required at the time of the collision response.

5.2.1.2 Recovering Grass Tiles

The grass tiles that have to be updated as well. As long as a grass tile has to recover and does not collide any longer it will be streamed through an additional recovering pass on the GPU. A recover time is assigned to each leaf node in order to know which of them needs to be recovered. This time is reset whenever a collision object intersects with the node or still is a neighbor of an affected node as described in the previous section. Consequently, the recovering time of the leaf nodes is decreased when no collision occurs. After the recovering time has elapsed it is estimated that each billboard which is covered by that leaf node has been recovered as well. A recovering grass tile is illustrated in Figure 5.4(a).

5.2.2 Updating Grass Tiles

The recovering and colliding grass tiles are updated via the stream output stage of the rendering pipeline. The GPU receives data on the input and writes the results to a location in the graphics memory. However, the input buffer has to be different from the output buffer (see Section 3.1.2). That is why the update of a grass tile involves a transfer of the data from one buffer to another. The buffer which contains the current grass tile is bound to the input assembler and the other buffer receives the updated data via the stream output stage. The grass tiles are processed through two steps as displayed in Figure 5.4(b). In each step the grass tiles are grouped to batches of greater size to minimize render calls as already mentioned in Section 4.2. All covered billboards are streamed to their corresponding location in the opposite buffer.

5.2.3 The Billboard's Collision Handling

If a grass tile is subject to an intersection, its billboards are updated by a collision handling process which performs different steps, as displayed in Figure 5.5. As long as no collision occurs, nothing is done. Whenever a collision becomes possible a refined mesh is applied to react to the collisions which results in a deformed billboard. If the collision is finished the billboard completely recovers to the undeformed shape. The transition of the unaffected mesh to the deformed mesh and finally returning to the undeformed mesh is made by performance considerations. Thus, the collision handling allows

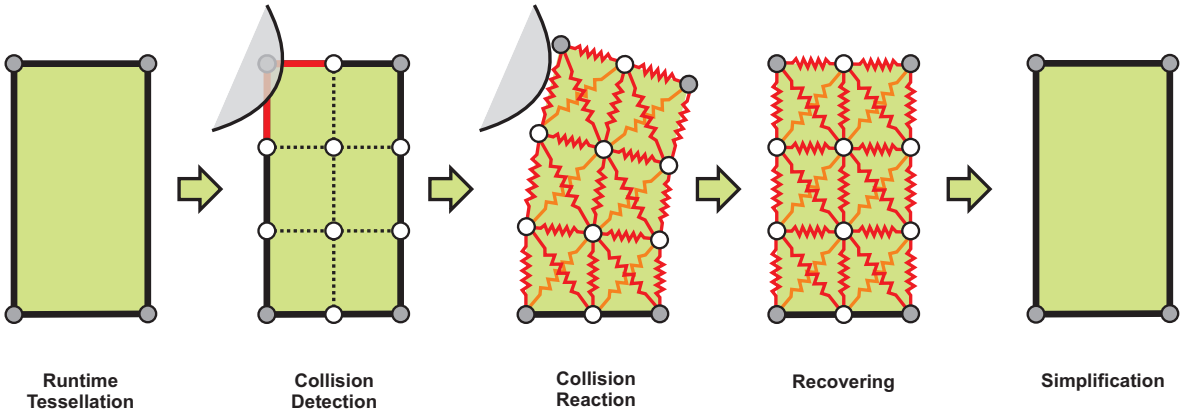


Figure 5.5: The billboard's states. Collision detection, reaction and recovering for a single billboard.

the grass to respond to collisions. It saves computation resources as recovered undeformed grass billboards are not as time consuming as the deformed billboards. Furthermore, the CPU does not have any information of the current billboards, thus, all these steps are processed by the GPU, which uses the geometry shader and the stream output stage. Moreover, it is necessary to have the ability to operate on a whole grass billboard at each invocation.

Two different types of grass tiles are handled. The tiles which are handled by the collision pass have to go through the whole collision handling process. Those tiles that are still have to recover are handled by a separate pass on the GPU. This avoids computation overhead for the recovering grass tiles which is caused by the collision tests. Only the parts which compute the current shape are executed in order to update the billboards (see Figure 5.3).

If collisions occur or the recovering of tiles is required, the quad mesh of the billboards is not sufficient for an adequate reaction. The current 3×4 vertices of the refined mesh is computed (see the first transition in Figure 5.5). Two cases are possible: On the one hand, if the billboard has not been involved in the collision process, the already wind animated quad is refined as described in Section 5.2.3.1. On the other hand, if the billboard still recovers, an interpolation among the vertices of the wind animated mesh and the vertices of the deformed mesh yields the current shape. The recovering process is outlined in Section 5.2.3.2. The GPU recovering pass ends at this point of the collision pipeline. Subsequently, the billboards are streamed to the graphics memory by using the stream output stage.

The next steps are applied for those billboards that might be affected by a collision object (see the collision handling and the collision response states in Figure 5.5). Please note the bright green box on the left in Figure 5.3. The collision test is split into two subsequent parts. A pre-test for each billboard shows if the grass billboard is influenced by a scene object. By testing distances among the bounding spheres of the grass billboard and of the scene object, several of the grass billboards can be excluded from subsequent handling as described in Section 5.2.3.3. The billboards, as long as no collision affects them, are streamed via the stream output stage without further handling. Thus, for

such obviously non-affected billboards only the wind animation and the recovering is applied during the collision handling.

Whenever the pre-test is passed an index is returned which specifies the collision object whose bounding sphere intersects the bounding sphere of the grass billboard. The refined mesh of the billboard is then tested for collisions with the depth cube of the indexed collider object. Each vertex of the billboard is tested separately, as described in detail in Section 5.2.3.4. If one of its vertices is found to be inside of the collision mesh, the vertex is translated along the surface normal of the mesh. This normal is stored in the depth cube in order to resolve the penetration as described in Section 5.2.3.5. However, moving each vertex of the billboard separately may lead to visually unpleasant distortions. Thereby, a cloth model based on spring constraints (see the collision reaction in Figure 5.5) is evaluated in order to preserve the overall shape of the clump of grass as described in Section 5.2.3.6.

Whenever a collision occurs, the recover time will be reset. As long as the billboard is penetrated it is not allowed to recover completely. The undeformed state which is solely affected by the wind animation is restored when the penetration has finished (see last transition in Figure 5.5). After the collision handling, the updated data of the billboard is streamed as a point primitive via the stream output stage. As a result, the tile's billboards exist in the buffer which is bound to the stream output stage, and thus, they are obsolete in the input buffer.

5.2.3.1 Refinement

```
inline
void Refine( inout float3 Vtx[VERTEX_COUNT] )
{
    float3 intVtx[2];
    float horVal, verVal;
    float horStep = 1.0/ float(VTX_CNT_HORIZONTAL-1);
    float verStep = 1.0/ float(VTX_CNT_VERTICAL-1);
    // refine mesh
    int idx = 0;
    for( float v=0; v<VTX_CNT_VERTICAL; v+=1.0 )
    {
        verVal = v * verStep;
        for( float h=0; h<VTX_CNT_HORIZONTAL; h+=1.0 )
        {
            horVal = h * horStep;
            //Interpolate among the horizontal edge
            intVtx[0] = lerp(Vtx[IDX_00], Vtx[IDX_02], horVal);
            intVtx[1] = lerp(Vtx[IDX_30], Vtx[IDX_32], horVal);
            //Interpolate among the vertical edge
            Vtx[idx++] = lerp(intVtx[0], intVtx[1], verVal);
        }
    }
}
```

Code Sample 3: The refinement. The positions of the mesh's inner vertices are interpolated bilinear among the mesh's edge vertices.

Each time a collision becomes possible (see Section 5.2.3.4) or the billboard still recovers (see

Section 5.2.3.2), the refinement mesh of the billboard is necessary. As the animation is a stateless process (see Section 4.4), it is possible to compute the original shape defined by the wind, without considering the current state which might be deformed by previous collisions. So, at first the wind animation described in Section 4.4.2 computes the current upper edge vertices. The resulting vertices $\mathbf{v}_{0,0}, \mathbf{v}_{0,2}$ and $\mathbf{v}_{3,0}, \mathbf{v}_{3,2}$ (see Equation 4.6) are bilinear interpolated to obtain the refined mesh:

$$\mathbf{v}_{i,j} = (1-\beta)((1-\alpha)\mathbf{v}_{0,0} + \alpha\mathbf{v}_{0,2}) + \beta((1-\alpha)\mathbf{v}_{3,0} + \alpha\mathbf{v}_{3,2}) \ ,$$

where $j \in \{0,1,2\}$ and $i \in \{1,...,3\}$. The blending terms $\alpha = j/2$ and $\beta = i/3$ preserve the network structure of the refined mesh. The interpolation is shown in Code Sample 3. As a result, the 3×4 mesh is obtained as needed throughout the subsequent stages of the collision handling.

5.2.3.2 Recovering

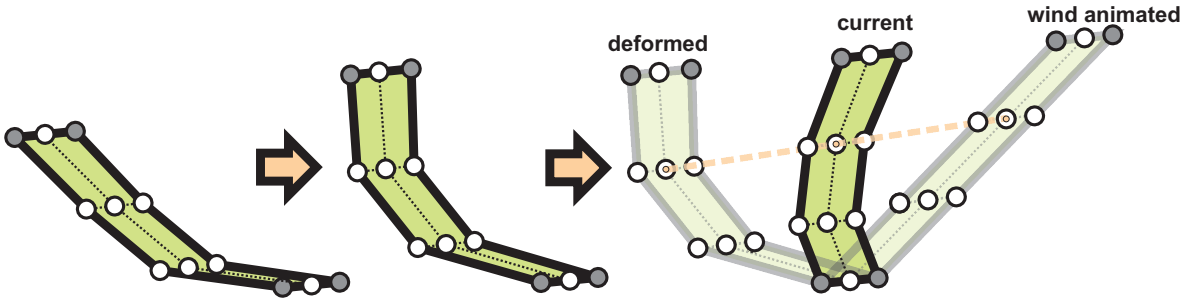


Figure 5.6: The billboard's recovering. The linear interpolation between the vertices of the deformed mesh and the vertices of the wind animated mesh results in the representation of the billboard with respect to the recover time left.

After a collision has occured, the grass billboard from now on will smoothly recover as long as there is recover time left. The refined mesh which is affected by the wind as well as the previously deformed mesh forms the basis of the recovering phase. The linear interpolation between the deformed vertices and their wind animated positions with respect to the recover time t_{rec} left results in the current shape of the grass clump (see Figure 5.6):

$$\mathbf{v}_{i,j} \leftarrow \ (1 - t_{\text{rec}}^3)\mathbf{w}_{i,j} \ + \ t_{\text{rec}}^3\mathbf{v}_{i,j} \ , \tag{5.1}$$

where $i \in \{1,...,3\}$ and $j \in \{0,...,2\}$. Furthermore, $\mathbf{w}_{i,j} \in \mathbb{R}^3$ are the vertices obtained by the previously described refinement step and $\mathbf{v}_{i,j} \in \mathbb{R}^3$ the last respectively the current recovered vertices of the billboard. The recover time $t_{\text{rec}} \in [0,1]$ is handled as a relative recover time in respect to a constant defined maximum time for recovering $t_{\max} \in \mathbb{R}$ which maps to the relative time 1. Instead of the linear decreased recover time the cubic function $t_{\text{rec}}^3 \in [0,1]$ is used for the interpolation in order to yield a smoother recovering which starts slow and then recovers fast. The recovering of a billboard is shown in Code Sample 4

```
inline
void Recover(
        inout float3 Vtx[VTX_CNT],
        in    float3 Wtx[VTX_CNT],
        in    float  RecTime )
{
    // Cubic transfer function to yield a smoother recovering
    float CubRecTime = RecTime * RecTime * RecTime;
    // Interpolate all vertices
    for( int idx=VTX_CNT_HORIZONTAL; idx<VTX_CNT; ++idx )
        Vtx[idx] = lerp(Vtx[idx],Wtx[idx], CubRecTime);
}
```

Code Sample 4: The recovering. A linear interpolation among the previously deformed mesh and the current subdivided mesh affected solely by the wind animation yields the current shape of the grass billboard in respect to the current remaining recover time. To smooth the recovering process a cubic transfer function is applied to the recover time.

After each recover step the recover time is linearly decreased with respect to the elapsed time t_{elap} which has passed since the last recovering process:

$$t_{\text{rec}} \leftarrow t_{\text{rec}} - \frac{t_{\text{elap}}}{t_{\max}} \tag{5.2}$$

If the recover time falls below zero and no collision occurs the animation of the grass billboard at that time will again be handled solely based upon the wind animation. The complete recovering is very important for the overall performance.

After each recovering step a collision test (see Section 5.2.3.4) with the current billboard's vertices $\mathbf{v}_{i,j}$ is made. As already mentioned, whenever a collision occurs the recover time of the affected billboard is reset to the previous defined recover time $t_{\max}$, which yields $t_{\text{rec}} = 1$.

5.2.3.3 Bounding Sphere Based Preclusion

Only for a small number of the grass billboards penetrations take place. A fast preclusion step prevents computation for the major part of the grass billboards. The bounding sphere test returns the index of the first object a collision is detected with. If no valid index is returned it is assumed that no penetration takes place.

Based on the current deformed or undeformed mesh of the billboard, the average position of the mesh's edge vertices is assumed to be the center $\mathbf{c}_{\text{bill}}$ of the sphere that encloses the current shape of the billboard:

$$\mathbf{c}_{\text{bill}} = \frac{1}{4}(\mathbf{v}_{0,0} + \mathbf{v}_{0,2} + \mathbf{v}_{3,0} + \mathbf{v}_{3,2}) \ .$$

Even if only the edge vertices are used without considering the inner vertices of the billboard, their mean point is accurate enough to test for collisions. Therefore, the length of the initial undeformed billboard $\|\mathbf{d}_{\text{grow}}\|$ is used as the sphere's radius. Although the billboard's length is an approximation

to the exact current bounding sphere's radius it still encloses the grass billboard as it is important. That is why in the undeformed state the span of the billboard is enlarged to its maximum length.

A billboard is not colliding if $(\|\mathbf{c}_{\text{bill}} - \mathbf{c}_{\text{obj}}\|)^2 > (\|\mathbf{d}_{\text{grow}}\| + r_{\text{obj}})^2$ [Eri04]. $\mathbf{c}_{\text{obj}}$ is the center and r_{obj} is the radius of the scene object's bounding sphere.

5.2.3.4 Collision Detection

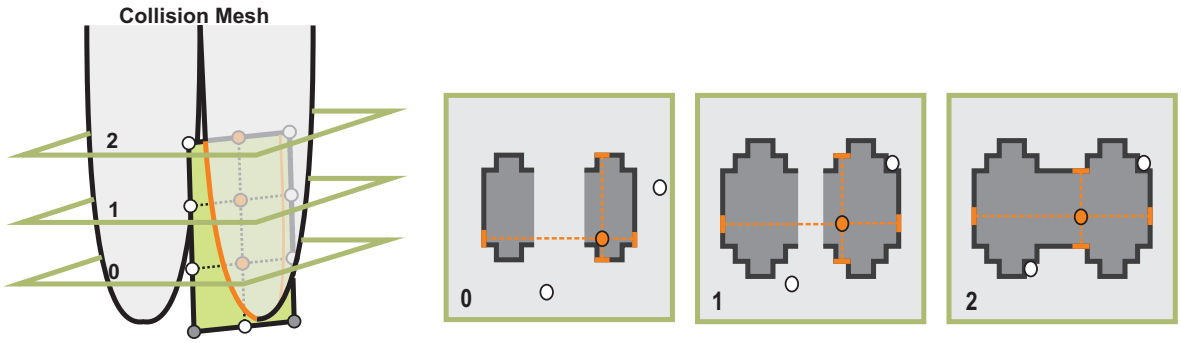


Figure 5.7: The depth cube based collision detection. For each of the orange inner vertices a collision is detected. (Left) The collision mesh penetrates the grass billboard. (Middle to Right) The depth tests for the billboard's vertices are shown in the depth map's projection space. Each of the three depth constellations represents the depth cube at a different height. Please note that only the orange vertices are occluded from all sides of the depth cube.

Since a collision is likely to come after the BS test is passed, the refined (deformed or undeformed) mesh of the billboard goes through a more exact collision test similar to the particle based detection described by [KLRS04]. Therefore, each vertex is tested separately.

A collision occurs if the vertex is occluded by the collision mesh along all six projection directions of the depth cube. The test for the penetration is performed in each of the six projection spaces. Thus, the vertex $\mathbf{v} = (v_x, v_y, v_z, 1)$ of the grass billboard is transformed:

$$\mathbf{v}' = \mathbf{v}\ \mathbf{T}_{WC \to DM}\ , \tag{5.3}$$

where $\mathbf{v}' = (v_x', v_y', v_z', 1)$ is the transformed vertex of the billboard. $\mathbf{T}_{WC \to DM}$ is a transformation from the world coordinate space to the projection space of the current distance map (see Section 5.1.2). The transformation is done for all the six distance maps.[1]

After the transformation the coordinate v_z' of the billboard's vertex is the distance to the projection plane of the current distance map. The coordinates $\mathbf{v}_x'$ and $\mathbf{v}_y'$ are used to look up the distance d into the distances map. The vertex is not penetrating along the projection direction if the sampled value d of the current distance map is greater than the relative distance v_z' (see appendix A.3 for a detailed

[1] A single transformation to the normalized view volume of the depth cube is possible as well. In that case the tests are performed in respect to the collider object's coordinate system [KLRS04].

```
uniform float4x4    DepthMapProj[6];
uniform float3      DepthCubeBoxSize;
Texture2DArray      DepthCube;

inline
bool TestCollision( inout float3 Vtx[VTX_CNT] )
{
    bool Intersects = false;
    float4 MapValue[6];
    float4 VtxProjSpace;
    // for each moveable vertex...
    for( int v=VTX_CNT_HORIZONTAL; v<VTX_CNT; ++v )
    {   // for each depth map...
        for( int m=0; m<6; ++m )
        {   // Transformation to the depth map's projection space.
            // Please note that the vertices of the billboard are
            // defined in world space coordinates
            VtxProjSpace = mul(float4(Vtx[v],1), DepthMapProj[m]);
            // Map the xy coordinate from [-1,1] to the range [0,1].
            VtxProjSpace.xy *= 0.5;
            VtxProjSpace.xy += float2( 0.5, 0.5 );
            // Sample the cube map to receive the depth value in
            // the w coordinate and the normal in the xyz coordinate.
            MapValue[m] =
                DepthCube.SampleLevel( NearestWrapSampler,
                                       float3(VtxProjSpace.xy,float(m)),
                                       0 ).xyzw;
            // Test the vertex to be closer to the near plane
            // as the object's shape. Therefore compute the
            // relative distance between both positions.
            MapValue[m].w = VtxProjSpace.z -  MapValue[m].w;
            if( MapValue[m].w < 0 )
                break; // no collision so break the loop here
        }
        // The vertex penetrates the object if all six depth maps
        // are yield the vertex to collide.
        if( m == 6 )
        {   // react to intersection
            Intersects = true;
            ResolveCollision(Vtx[v], DepthCubeBoxSize, MapValue);
        }
    }
    return Intersects;
}
```

Code Sample 5: The collision detection. The depth cube based collision detection of the billboard. The HLSL code fragment shows the collision handling for the billboard's mesh. The collision test is based upon a distance comparison between the transformed vertex and the distance which is looked up into the distance maps. For each of the scene objects a separate method is implemented as the dynamic assignment of texture slots to samplers at runtime is not supported by the GPU.

description of distance comparisons). If so, the other distance maps are not tested. An collision occurs if all distance values are smaller than $v_z^{'}$. In formal terms the following computations are made:

$$r = v_z^{'} - d \, , \tag{5.4}$$

where $r \in [0, 1]$ is the distance between the collision mesh's shape and the billboard vertex $\mathbf{v}$ in the distance map's projection space. If $r > 0$ the vertex is occluded along the projection direction. The

vertex collides if all distance maps m report the vertex to be occluded as shown in Figure 5.7:

$$r_m > 0 \ \ \forall m \ , \tag{5.5}$$

where r_m is the result of Equation 5.4 for the mth distance map DM_m , $m = 0, ..., 5$.

The Equation 5.5 is applied for each vertex of the billboard. For each of the collider objects the six depth cube transformations as well as the depth cube texture are passed to the shader. As texture slots are not interchangeable after the compilation of the shader code the collision detection is divided into separate function calls, one for each scene object. The index which is returned by the bounding sphere test is used to identify the correct branch. The shader fragment concerning the depth test for a single object is outlined in Code Sample 5.

5.2.3.5 Resolving Collisions

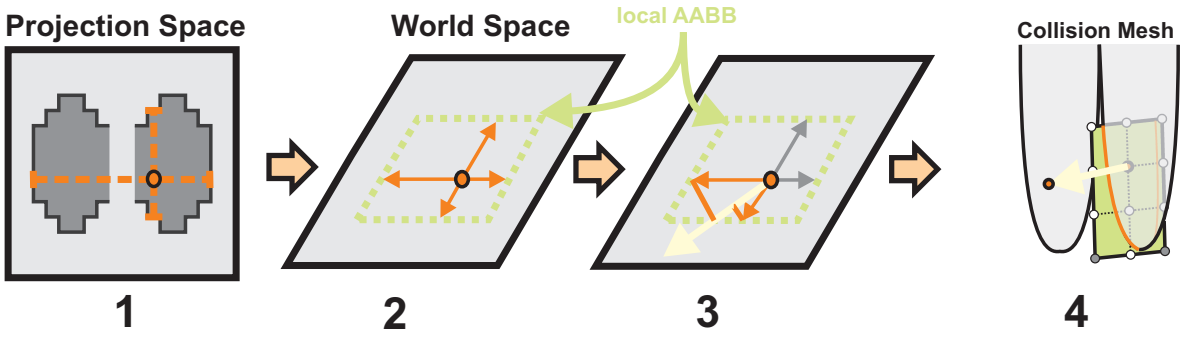


Figure 5.8: The collision response. First the distances of the projection space are transformed to world space distances which form the local AABB of the collision mesh. The reaction direction is looked up into that distance map which stores the smallest distance between the vertex and the collision mesh's surface. Then, all world space distances along the reaction direction are summed up in order to yield the reaction strength. Finally, the vertex is translated along the reaction direction.

If a collision has been detected for a vertex of the grass billboard, its position is moved in the direction of the shortest way out of the object's shape. The direction of the reaction depends on the normal information which is stored in the distance maps of the depth cube [KLRS04]. In addition, the reaction strength is obtained in respect to the local bounding box of the vertex. The local bounding box is formed by the six distances along each of the world space axis an thus along each of the distance maps projection directions $\mathbf{d}_m$.

At first, the relative distances $r_m \in [0, 1]$, $m = 0, ..., 5$ (see Equation 5.4) are weighted with the size $\mathbf{s} = (s_x, s_y, s_z) \in \mathbb{R}^3$ of the depth cube's axis aligned bounding box in order to map them to their corresponding world space distances:

$$w_m = r_m \cdot (\mathbf{d}_m \bullet \mathbf{s}) \ , \tag{5.6}$$

where w_m is the world space distance from the vertex to the collision mesh's surface in respect to

the projection direction $\mathbf{d}_m$, $m = \{0, ..., 5\}$. Those values then form the local axis aligned bounding box of the collision mesh in respect to the position of the billboard's vertex $\mathbf{v}$. Please note the second image of Figure 5.8.

The vertex is translated along the normal vector $\mathbf{n}_{react} \in \mathbb{R}^3$ of the surface. Therefore, the normal information of the distance map k is used which provides the shortest distance of the vertex to the collision mesh's surface:

$$w_k \leq w_m \; \forall \; m \in \{0, ..., 5\} \wedge m \neq k \; . \tag{5.7}$$

```
inline
void ResolveCollision( inout float3   Vtx,
                       in    float3   BoxSize,
                       in    float4   MapValue[6] )
{
    // Find the minimal distance between the vertex and
    // the collision mesh's surface in order to receive the
    // normal which directs along the shortest way out of the
    // object.
    float WorldDist[6];
    WorldDist[0] = (MapValue[0].w * BoxSize.x);
    WorldDist[1] = (MapValue[1].w * BoxSize.x);
    WorldDist[2] = (MapValue[2].w * BoxSize.y);
    WorldDist[3] = (MapValue[3].w * BoxSize.y);
    WorldDist[4] = (MapValue[4].w * BoxSize.z);
    WorldDist[5] = (MapValue[5].w * BoxSize.z);

    int NormIdx = 0;
    for( int m=1; m<6; ++m )
        if( WorldDist[m] < WorldDist[NormIdx] )
            NormIdx = m;

    // Compute the reaction strength. Therefore sum up the
    // distances along the normal with respect to
    // the six axis aligned projection directions.
    float ReactStrength = 0.0;
    ReactStrength += max( WorldDist[0]  * MapValue[NormIdx].x, 0);
    ReactStrength += max( -WorldDist[1] * MapValue[NormIdx].x, 0);
    ReactStrength += max( WorldDist[2]  * MapValue[NormIdx].y, 0);
    ReactStrength += max( -WorldDist[3] * MapValue[NormIdx].y, 0);
    ReactStrength += max( WorldDist[4]  * MapValue[NormIdx].z, 0);
    ReactStrength += max( -WorldDist[5] * MapValue[NormIdx].z, 0);
    ReactStrength *= 0.5;

    // Translate vertex along the surface normal with respect to
    // the reaction strength.
    Vtx += MapValue[NormIdx].xyz * ReactStrength;
}
```

Code Sample 6: The collision response. The collision reaction for each vertex. First the shortest of the world space distances is found and then the reaction strength is computed in order to translate the vertex in respect to the surface normal.

In simple terms, the distance map k is identified by the distance which is smaller than the distances obtained by the other maps m. The normal $\mathbf{n}_{\text{react}}$ of the distance map k is looked up by using the pixel coordinates (v_x', v_y') of the vertex in the distance map's projection space which were computed

during the collision detection.

As the normals all have unit length (see Section 5.1.2.2), the reaction strength which means the magnitude of the translation is evaluated, as also illustrated in the third step in Figure 5.8:

$$s_{\text{react}} = \sum_{m=0}^{5} \max(\mathbf{n}_{\text{react}} \bullet (\mathbf{d}_m \cdot w_m), 0) \ . \tag{5.8}$$

The reaction strength $s_{\text{react}} \in \mathbb{R}$ is the total of all the world space distances projected to the surface normal. As opposite directed vectors yield negative values they are precluded from the computation. For this purpose the $\max()$ function prevents the reaction strength from being influenced by negative distances that are directed contrary to the surface normal.

Finally, the vertex $\mathbf{v}$ is translated along the surface normal with respect to the reaction strength (see function 5.8) as shown in the last step of Figure 5.8:

$$\mathbf{v} \leftarrow \mathbf{v} + s_{\text{react}} \ \mathbf{n}_{\text{react}} \ . \tag{5.9}$$

However, the computation does not translate the vertex to the edge of the local bounding box since no quadratic function of the world space distances is applied in function 5.8. Instead of this it is just guaranteed that the vertex is translated out of the local bounding box as it is intended. The information of the collision response is shown in Code Sample 6.

5.2.3.6 Preserving the Grass Shape

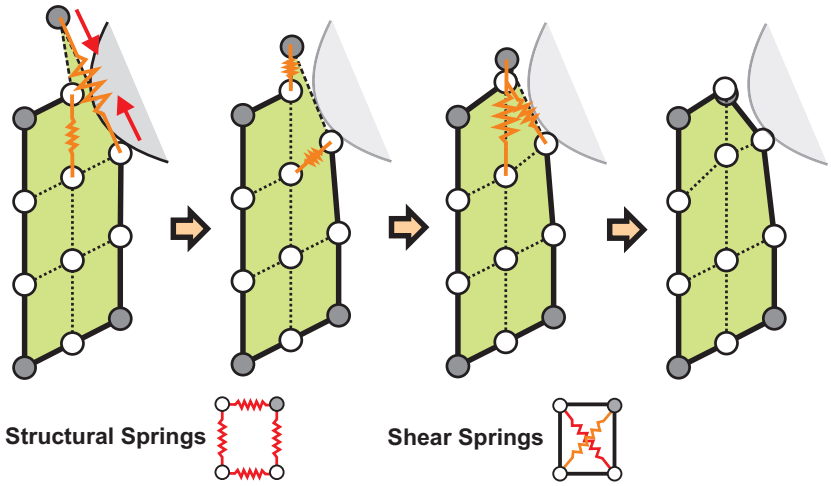


Figure 5.9: The spring relaxation. After a collision response two types of springs are used to preserve the distances between linked vertices. In Figure 5.9 a single recursion is shown for the upper right quad which has been distorted. Please note that if the reponsiveness of the vertices are equal each spring translates both of the connected vertices along the direction of their linkage.

Since the translation of the vertices is done separately, the shape of the grass has to be taken into

consideration. Therefore, a post correction step is applied. This step preserves the overall shape of the grass billboard [FGL03, Zel07]. In contrast to the previous collision handling steps, this correction step accounts the billboard as a network of 3×4 vertices. Moreover, the network is expanded by a set of virtual linear springs. Each of the springs takes care of the distance between two vertices. Whenever such a spring is compressed or stretched, which means the connected vertices diverge or converge, the resulting spring force affects their vertices. The vertices are moved along their connected direction in respect to the spring's stiffness.

Thus, the spring force $\mathbf{f}_{i,j,k,l} \in \mathbb{R}^3$ between two vertices $\mathbf{v}_{i,j}$ and $\mathbf{v}_{k,l}$ with $i, k \in \{0, ..., 3\} \wedge j, l \in \{0, ..., 2\}$ is defined as:

$$\mathbf{f}_{i,j,k,l} = k_{i,j,k,l}(\|\mathbf{d}_{i,j,k,l}\| - d^0_{i,j,k,l}) \; \frac{\mathbf{d}_{i,j,k,l}}{\|\mathbf{d}_{i,j,k,l}\|} \; , \tag{5.10}$$

where $\mathbf{d}_{i,j,k,l} = \mathbf{v}_{k,l} - \mathbf{v}_{i,j}$ is the direction of the connection between both vertices, $d^0_{i,j,k,l}$ is the natural length between the both billboard vertices. The natural length depends on the type of spring. Furthermore, $k_{i,j,k,l} \in [0, 1]$ is the stiffness of the spring. A value of 1 results in a conservative spring in contrast to a value of 0 which has no effect.

The evolution of the force between the two vertices that have been connected by a spring results in a translation for each of the vertices. The linear spring constraints are directly applied to each of the two connected vertices [Zel07] instead of summing up all spring forces for each vertex and finally adjusting the vertex position [Pro95]. The formula for the adjustment of both vertex positions is:

$$\begin{aligned} \mathbf{v}_{i,j} &\leftarrow \mathbf{v}_{i,j} + r_{i,j} \; \mathbf{f}_{i,j,k,l} \quad \text{and} \\ \mathbf{v}_{k,l} &\leftarrow \mathbf{v}_{k,l} - r_{k,l} \; \mathbf{f}_{i,j,k,l} \quad , \end{aligned}$$

where $r_{i,j}$ is the responsiveness for vertex $\mathbf{v}_{i,j}$ and $r_{k,l}$ is the responsiveness for vertex $\mathbf{v}_{k,l}$. The responsiveness is added in order to distinguish between fixed and movable vertices. In addition, the restriction $r_{i,j} + r_{k,l} = 1$ is made to preserve the strength of the spring force. As a fixed vertex should not be moved the responsiveness is set to zero, whereas the movable vertex then is completely responsive. If both vertices are not fixed they have the same responsiveness and thus, $r_{i,j} = r_{k,l} = 0.5$. An example of this type of constrained based forces is given in Code Sample 7

As the linkage through springs is built up between neighboring vertices, two types of springs are applied for that purpose. Referring to [Pro95], the first type is a so-called *structural spring* which takes care of the compression and stretching of the grass billboard. In formal terms, the vertices $\mathbf{v}_{i,j}$ and $\mathbf{v}_{i+1,j}$ are connected with a vertical *structural spring* and the vertices $\mathbf{v}_{i,j}$ and $\mathbf{v}_{i,j+1}$ are connected by a horizontal structural spring. Their natural length is $d^0_{i,j,k,l} = s_0$ for the horizontal aligned spring and is $d^0_{i,j,k,l} = s_1$ for the vertical aligned spring. s_0 is the width and s_1 is the height of the billboard's refined quads. Both lengths in addition with a diagonal length s_2 are stored during the procedural generation process (see Section 4.3). As the ground vertices $\mathbf{v}_{0,j}$ are fixed the horizontal structural springs are not applied among them. The second type is called *shear spring* and takes care

```
inline
void LenConstraint( inout float3 Vtx0,
                    in    float  Resp0,
                    inout float3 Vtx1,
                    in    float  SpringLen,
                    in    float  SpringStiff )
{
    // Compute the distance information
    float3 DistVec = Vtx1 - Vtx0;
    float Distance = length(DistVec);
    // Compute the spring force
    float3 SpringForce =
       (SpringStiff * (Distance - SpringLen) * (DistVec/Distance));
    // Apply the spring force
    Vtx0 += Resp0 * SpringForce;
    Vtx1 -= (1-Resp0) * SpringForce;
}
```

Code Sample 7: The length constraints. The length constraint function preserves the distances between two vertices with respect to the stiffness of a spring and the stress which occurs to the spring. The produced stress then is solved by translating both vertices.

of shear stresses affecting the grass billboards. These springs link the vertices $\mathbf{v}_{i,j}$ and $\mathbf{v}_{i+1,j+1}$ and the vertices $\mathbf{v}_{i+1,j}$ and $\mathbf{v}_{i,j+1}$. Their natural length is equal to the diagonal length s_2 of the refined billboard quad. Both types of springs are shown in Figure 5.9.

Since the execution of one spring force affects the neighbouring springs as well, more iterations over all springs have to be applied to get a good result due to the strong effect of stiff springs. For less stiff springs a single iteration also yields visually pleasant results due to the small number of vertices. The implementation is shown in Code Sample 8.

```
uniform float3 SpringStiff;

void SatisfySpringConstraints( inout float3  Vtx[VTX_CNT],
                               in    float3  SpringLens )
{
    for( int r=0; r<NUM_RECURSIONS; ++r )
    {
        // 1. satisfy the vertical structural springs of the fixed
        //    ground vertices
        LenConstraint(Vtx[IDX_00],0,Vtx[VTX_CNT_HORIZONTAL],SpringLens.x,SpringStiff.x);
        LenConstraint(Vtx[IDX_01],0,Vtx[VTX_CNT_HORIZONTAL+1],SpringLens.x,SpringStiff.x);
        LenConstraint(Vtx[IDX_02],0,Vtx[VTX_CNT_HORIZONTAL+2],SpringLens.x,SpringStiff.x);
        // 2. satisfy the shear springs of the fixed ground vertices
        LenConstraint(Vtx[IDX_00],0,Vtx[VTX_CNT_HORIZONTAL+1],SpringLens.z,SpringStiff.z);
        LenConstraint(Vtx[IDX_01],0,Vtx[VTX_CNT_HORIZONTAL],SpringLens.z,SpringStiff.z);
        LenConstraint(Vtx[IDX_01],0,Vtx[VTX_CNT_HORIZONTAL+2],SpringLens.z,SpringStiff.z );
        LenConstraint(Vtx[IDX_02],0,Vtx[VTX_CNT_HORIZONTAL+1],SpringLens.z,SpringStiff.z );
        // 3. satisfy the vertical structural springs of the movable
        //    vertices
        int v, h;
        int curIdx;
        for( v=1; v<(VTX_CNT_VERTICAL-1); ++v )
        {
            for( h=0; h<VTX_CNT_HORIZONTAL; ++h )
            {
                curIdx = (v*VTX_CNT_HORIZONTAL)+h;
                LenConstraint( Vtx[curIdx],0.5,Vtx[curIdx+VTX_CNT_HORIZONTAL],
                               SpringLens.x,SpringStiff.x);
            }
        }
        // 4. satisfy the horizontal structural springs of the movable
        //    vertices
        // ...
        // 5. satisfy the shear springs of the movable vertices
        // ...
    }
}
```

Code Sample 8: The spring network. The HLSL code shows how the vertex-spring network is satisfied in order to preserve the grass shape after collisions. First the springs of the fixed ground vertices are resolved. Afterwards the springs among the movable vertices are relaxed. The stiffness **SpringStiff** of the springs therefore is passed as a constant to the shader. Note that the stiffer the springs the more recursions over all springs have to be executed because each spring affects neighboring springs as well.

Chapter 6

The Rendering System

Rendering of natural sceneries is a complicated subject especially if realistic global illumination is applied. In addition the process should take as little time as possible and offer a high degree of detail. Nevertheless, lighting quality has a high influence on the performance. Many techniques are often working together to create a convincing illusion of grassy fields, for example shadow mapping, local illumination model and screen door transparency [Wha05]. The rendering of natural environments respectively grass turns out to be an arrangement of approximations to physical phenomena. The rendering dealt with in this chapter applies a more physically based global illumination technique to the grass layer. However, the approximation used is still coarse: A pre-process computes irradiance information affected by static scene objects. At runtime this information is used to illuminate each vertex of the two sided grass billboards. Finally, the alpha channel of the semi-transparent decal texture which covers the billboard is giving the clump of grass its correct shape.

6.1 The Billboard's Rendering Equation

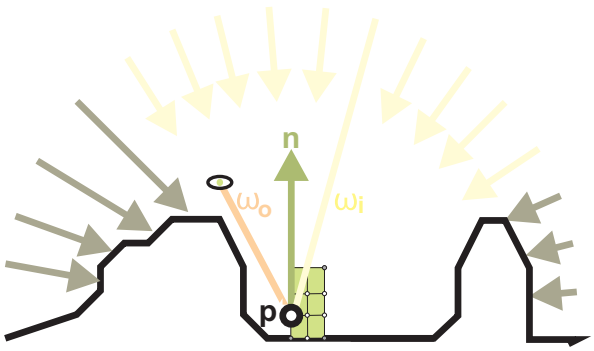


Figure 6.1: The global illumination. The rendering equation is approximated for each vertex of the billboard. Scene geometry absorbs some of the environmental light. As a result, smooth shadows are cast on the grass billboards. The light that reaches the billboards is reflected to the viewer.

For real-time rendering local illumination based upon the object's surface is often applied. Effects like transparency, reflections and shadows caused by other objects are usually not covered. These effects come with their costs. Global illumination which is applied among scene objects is computationally expensive. Thus, objects are lit by approaches which approximate global environmental lighting in order to improve the realism of the scene. Such a technique is used to illuminate the grass billboards.

Before the detailed rendering techniques are handled, the rendering equation which is approximated by the render process is explained in short. The equation is derived from the global illumination model introduced by [GTGB84] and [Kaj86]:

$$\mathbf{L}_o(\mathbf{p}, \omega_o) = \int_{\mathbf{\Omega}} \rho(\mathbf{p}, \omega_i, \omega_o)\ \mathbf{g}(\mathbf{p}, \omega_i)\ \mathbf{L}_i(\mathbf{p}, \omega_i)\ d\omega_i\ , \tag{6.1}$$

where $\mathbf{L}_o(\mathbf{p}, \omega_o)$ is the radiance at point $\mathbf{p}$ traveling in direction ω_o. $\mathbf{L}_i(\mathbf{p}, \omega_i)$ is the light directed along ω_i to point $\mathbf{p}$. $\rho(\mathbf{p}, \omega_i, \omega_o)$ is the reflectance term defining the relationship between incoming and outgoing light relating to the surface at point $\mathbf{p}$. Finally, the occlusion term $\mathbf{g}(\mathbf{p}, \omega_i) \in \{0, 1\}$ defines whether or not the point is reachable for the incoming light $\mathbf{L}_i(\mathbf{p}, \omega_i)$. When it is not obscured the term returns one and otherwise zero. The Equation 6.1 computes the energy flow between the surfaces of all scene objects. In other words, the total amount of incoming light intensity at a point of a surface depends upon the sum of all light intensity reaching that point from other surfaces.

Besides, the equation applied to the grass billboards is less complex and does not account for indirect light transport. As a consequence the incoming light is always environmental lighting. As the environment is assumed to be far away, the incoming light for each location inside the grass layer only depends on the direction. Therefore $\mathbf{L}_i(\mathbf{p}, \omega_i)$ is rewritten as $\mathbf{L}_{env}(\omega_i)$ which is the environmental light coming from direction ω_i. However, the equation still covers the second major aspect of global illumination. That is the occlusion of the environment caused by scene geometry:

$$\mathbf{L}_o(\mathbf{p}, \omega_o) = \int_{\mathbf{\Omega}} \rho(\mathbf{p}, \omega_i, \omega_o)\ \mathbf{g}(\mathbf{p}, \omega_i)\ \mathbf{L}_{env}(\omega_i)\ d\omega_i\ . \tag{6.2}$$

The influence caused by other objects is reduced to the occlusion term as can be seen in Figure 6.1. In a pre-process the irradiance is integrated for a fixed set of points within the grass layer's bounding volume (see Section 6.2). The irradiance is defined as:

$$\mathbf{E}(\mathbf{p}) = \int_{\mathbf{\Omega}} \mathbf{g}(\mathbf{p}, \omega_i)\ \mathbf{L}_{env}(\omega_i)\ d\omega_i\ . \tag{6.3}$$

The only term which is missing compared with Equation 6.2 is the reflectance term which cannot be pre-computed as the vertex normal is required for the reflectance computation. During runtime Equation 6.2 is approximated for the billboard's vertices by sampling the pre-computed irradiance and computing the reflectance as described in Section 6.3.2.

6.2 The Irradiance Volume

As the integration of Equation 6.2 on the basis of polygonal objects cannot be done in real-time, the parts which account for interobject occlusion must be pre-processed. Therefore, dynamic global illumination throughout the grass layer is achieved by pre-computing irradiance information within a volume. This approach is similar to the ambient occlusion volume presented by [CL07]. The irradiance for each voxel is determined by sampling an environment map with the aid of additional ambient occlusion information. The captured irradiance samples are stored in two texture arrays. Shifting the irradiance computation into a relatively expensive pre-processing allows fast dynamic global illumination. The irradiance volume is used to dynamically shade the grass billboards in a static scene at runtime.

6.2.1 Volume Set-Up

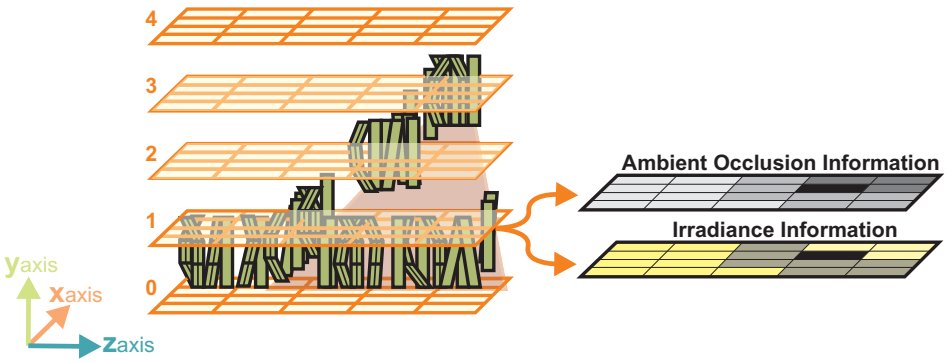


Figure 6.2: The setup of the irradiance volume. The scene is covered by 5 texture slices along the y axis. Each slice stores ambient occlusion quantities as well as irradiance information for each voxel.

The irradiance volume covers the whole grass layer. Therefore it has the same extent as the grass layer's bounding volume. The volumetric data is distributed in two texture arrays: The ambient occlusion quantities and the irradiance information are stored in different textures. Each array contains $n_{tex} \in \mathbb{N}$ layers (with the restriction that $n_{tex} < 1024$ as described in Section 3.2). Each of the two dimensional layers summarizes a slice of voxels in the xz plane of the world coordinate system. The slices are oriented along the world space y axis as is shown in Figure 6.2. The number of slices n_{tex} as well as the resolution of the volume has an influence on the accuracy of the dynamically interpolated data.

6.2.2 Ambient Occlusion Information

Ambient occlusion indicates how much of the positive hemisphere is occluded at a point in the scene. Additionally, the average incoming light direction is provided. The information of visibility is obtained by employing hardware-based shadow maps. Each of these shadow maps is generated in re-

spect to a random light direction pointing to the hemisphere above the grass layer. The iteration over all shadow maps respectively light directions finally results in the accumulated ambient occlusion information stored for each voxel of the volume.

6.2.2.1 Occlusion Term

The occlusion term $\mathbf{g}(\mathbf{p}, \omega_i)$, being a part of Equation 6.2, returns whether a point in the scene is occluded along a direction ω_i or not. Conceptually, in order to test for occlusion, a ray starts at the point $\mathbf{p} \in \mathbb{R}^3$ for which the occlusion has to be determined and is directed to the positive hemisphere above the grass layer. If the ray hits a static scene object the direction is occluded and as a consequence $\mathbf{g}(\mathbf{p}, \omega_i)$ returns the value zero. If no object is hit, the returned value is one:

$$\mathbf{g}(\mathbf{p}, \omega_i) = \begin{cases} 0 & \text{if } \mathbf{p} \text{ occluded along } \omega_i \\ 1 & \text{else} \end{cases} \tag{6.4}$$

Casting a ray for each point respectively voxel inside the volume is too expensive. Therefore GPU based distance maps respectively shadow maps [Wil78] are employed [PG04]. Such a distance map stores distances of all shadow emitters with respect of the light direction. The distance of the shadow emitters is received as a consequence of the transformation of the object to the distance map's projection space. The projection space is determined by the projection direction ω_i and the extent of the scene's axis aligned bounding box. So the distance map is covering the whole scene. This enables to determine occlusion for all voxels of the ambient occlusion volume without recomputing the distance map for each voxel: A distance comparison in the distance map's projection space is used to determine if a voxel is occluded or not (see appendix A for a more detailed description of distance maps).

Distance maps therefore are very efficient to test occlusion along a direction. It should be remarked that the resolution of the maps and the grass layer's extremities determine the accuracy of the test.

6.2.2.2 Occlusion Quantities

Ambient occlusion is a technique which is often used for real-time environment lighting of diffuse surfaces[Lan02, PG04]. In general the ambient occlusion data of a point provides two quantities: First it supplies how much of the environment is visible at that point. This information is called the accessibility $a \in [0, 1]$ of a point. Second, the average direction of incoming light so called bend normal $\mathbf{b} \in \mathbb{R}^3$ is provided [PG04]. Both are used to approximate global illumination effects that can otherwise only be attained by costly computations.

The ambient occlusion quantities for the volume are obtained by an iterative process over all light directions. Each light direction is randomly oriented towards the positive hemisphere above the grass layer. The key part of the computation is the visibility test $\mathbf{g}(\mathbf{p}, \omega_i)$. Therefore, a distance map for each light direction $\omega_i \in \mathbb{R}^3$ is employed (see the previous Section 6.2.2.1). After projecting all

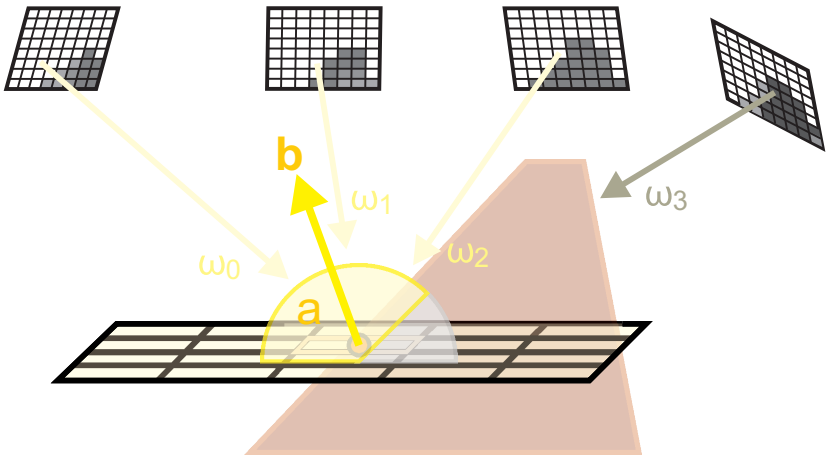


Figure 6.3: The ambient occlusion information. Each voxel of the irradiance volume is tested to be visible along each light direction. The occlusion test is based upon distance maps. As a result, the average unoccluded light direction $\mathbf{b}$ is determined for each voxel. In addition, the quantity of unoccluded directions a is retrieved.

scene objects a lookup with respect to the voxel position determines if the voxel is obscured or not along the projection direction (see appendix A.3). Whenever the direction is unoccluded it affects the bend normal as well as the accessibility value for that voxel. More formal, for each voxel at position $\mathbf{p} \in \mathbb{R}^3$ the accessibility is averaged over all directions:

$$a = \frac{1}{n} \sum_{i=1}^{n} \mathbf{g}(\mathbf{p}, \omega_i) \tag{6.5}$$

where n is the number of random directions and $\mathbf{g}(\mathbf{p}, \omega_i)$ is the distance map based occlusion term from Equation 6.4. In addition, for each voxel the bent normal is calculated:

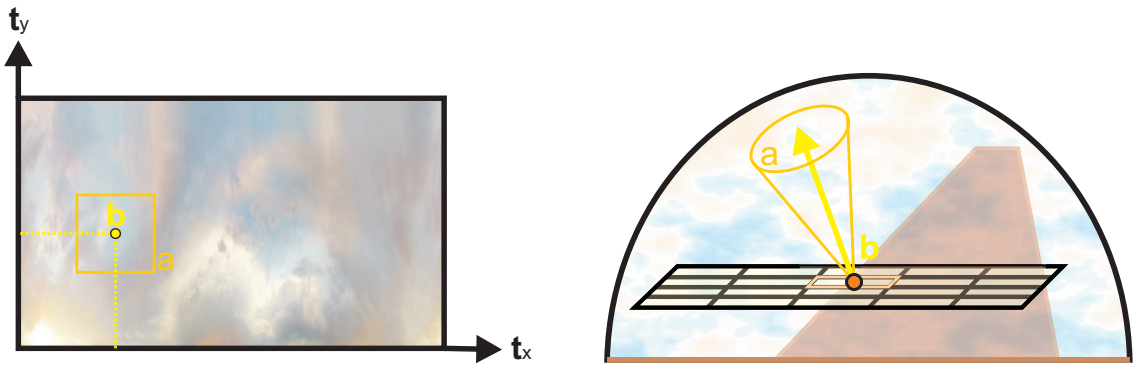
$$\mathbf{b} = \sum_{i=1}^{n} \mathbf{g}(\mathbf{p}, \omega_i) \cdot \omega_i \tag{6.6}$$

The distance maps, each of which summarizes occlusion information for the scene along a random direction, are iteratively applied to the voxels of the volume. This process is displayed in Figure 6.3. As only eight render targets can be bound in one single render call, the volume is refined in batches of eight slices. Each of these batches then is updated separately by a pixel shader. As a result for each voxel of the volume the average amount of the unoccluded area with respect to the positive hemisphere and the bent normal is computed.

The objects serving as occluders must be static. Whenever the constellation between them and the grass layer is modified the ambient occlusion quantities have to be recomputed for the whole volume.[1]

[1] Another useful approach is described by [CL07]: The occlusion information is stored by using a spatial tree structure over the scene. The ambient occlusion quantities are dynamically recomputed whenever a scene object has moved out of a tree's node.

6.2.3 Irradiance Information



(a) (b)

Figure 6.4: The irradiance information. The hemisphere above the grass layer is covered by an environment map. The map is sampled by employing a spherical parameterization of the bent normal $\mathbf{b}$. The number of unoccluded light directions a determines the sampled area of the map (see Figure 6.4(a)). Both quantities define the cone of incoming light for each voxel of the irradiance volume (see Figure 6.4(b)).

For each voxel of the volume the incoming irradiance is approximated by using the ambient occlusion information of the voxel. As mentioned in Section 6.1, the irradiance is reduced to environmental lighting. Therefore, $\mathbf{L}_{env}(\omega_i)$ is represented by a single environment map which covers the whole hemisphere over the grass layer (see Figure 6.4). A blurred lookup from the environment map approximates the irradiance for a point [PG04] within the bounding volume of the grass layer. This results in a coarse approximation to the irradiance function 6.3 but it is evaluated much faster and can be computed on modern graphics hardware. The ambient occlusion quantities of each voxel determines the irradiance $\mathbf{E}$ (see Figure 6.4(b)):

$$\mathbf{E}(\mathbf{b}, a) = a \cdot \text{env}(\mathbf{t}, dt_x, dt_y) \ , \tag{6.7}$$

where $dt_x = dt_y = a^4 \in [0, 1]$ are the derivatives in each direction of the texture. The derivatives are determining the area respectively the mip-map level of the environment map in which texture filtering occurs (see Figure 6.4(a)). The greater the derivatives are, the higher is the interpolated mip-map level and the more blurred is the returned value. In addition, the sampled value is multiplied by the accessibility a in order to darken areas that are more occluded than others. The derivative based sampling function env() bi-linearely interpolates among the texels at coordinates $\mathbf{t} \in [0, 1]^2$ of the mip-map. The environment map is defined in the latitude longitude format. The bent normal $\mathbf{b} = (b_x, b_y, b_z) \in \mathbb{R}^3$ is used to address a point in the environment map (see Figure 6.4(a)). The mapping to the environment coordinate space is handled by spherical parameterization:

$$\mathbf{t} = \left(\frac{1}{2\pi}(\arctan(\frac{b_y}{b_x}) + \pi),\ \frac{1}{\pi}\arccos(b_z) \right) . \tag{6.8}$$

Similary to the creation of the ambient occlusion information a pixel shader computes eight texture levels of the irradiance volume in one single render call. The precomputed irradiance does not account for dynamic lighting environments. In that case the computation of the irradiance information has to be done during runtime as described by [PG04]. However, this causes far more computations for each fragment of the grass billboards.

6.3 The Rendering Process

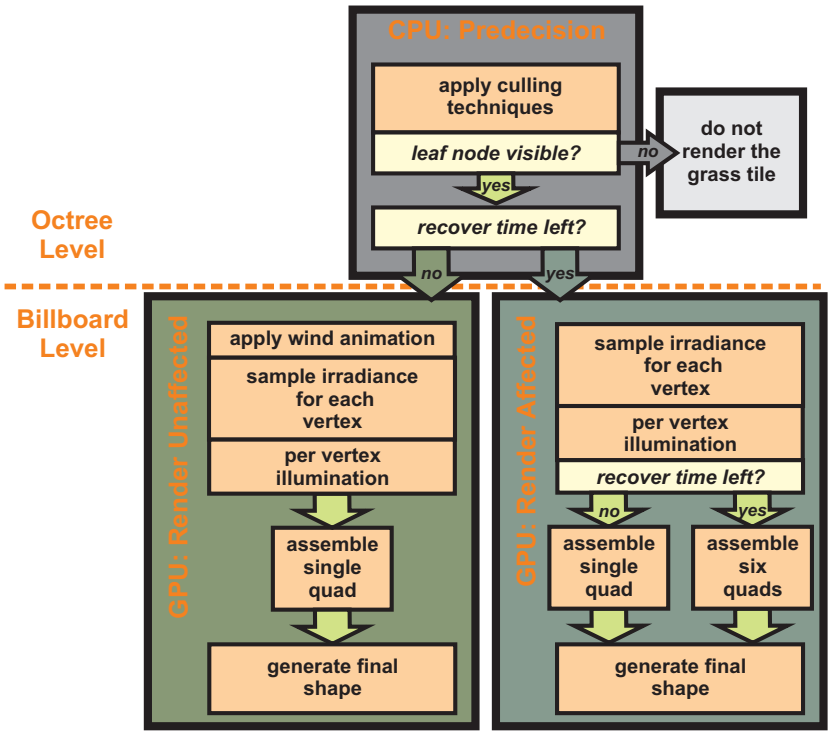
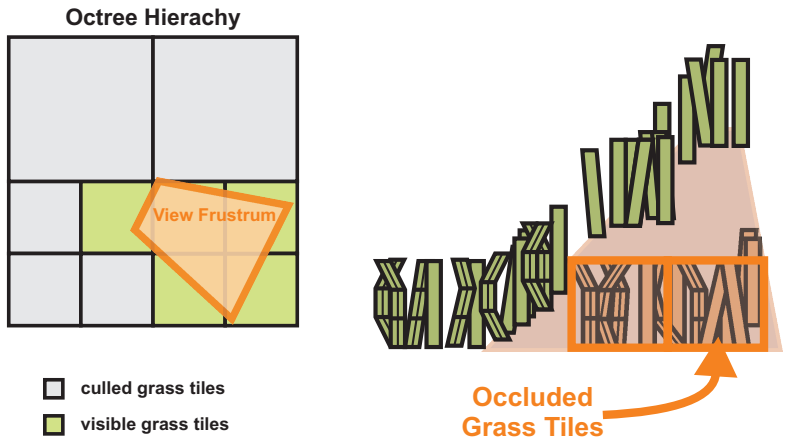


Figure 6.5: The render process. The process is divided into two passes in order to speed up rendering for those grass tiles that are not affected. Beforehand invisible clusters are culled.

The rendering process is split into two separate render passes with regard to the outcome of the collision pipeline as can be seen in Figure 6.5. Some grass tiles might have been affected by collisions while others have not been updated yet. Before rendering a culling of invisible grass tiles takes place. The main difference between both render passes is that the billboards which have not been updated by the collision system still have to be animated during the rendering. The grass tiles which are updated by the collision pipeline might contain deformed billboards. Those deformed billboards go through a finer tessellation.

6.3.1 Culling Grass Tiles



(a) (b)

Figure 6.6: The culling techniques. Each node which is not covered by the view-frustrum (see Figure 6.6(a)) or is occluded by other geometry (see Figure 6.6(b)) is culled from rendering.

Often only a small portion of the grass layer is visible. The majority of grass billboards can be omitted from rendering. Two culling techniques are applied on the CPU as is displayed in Figure 6.6. The visible set of grass tiles is determined by an intersection calculation based on the bounding boxes of the hierarchical octree structure and the view frustrum. Afterwards, the viewed leaf nodes are further tested for occlusion by employing GPU based occlusion queries. After both culling techniques have been applied the remaining set of grass tiles is rendered.

6.3.1.1 Viewport Culling

As the grass clusters which are outside of the view-frustrum are clipped during the rasterizer stage of the rendering pipeline (see Section 3.1), they can be culled beforehand [MH99]. The occlusion test is based upon the hierarchical octree structure of the grass layer. As each of the nodes encloses its children, the tree can be efficiently traversed in order to cull the invisible nodes. Starting at the root node all eight child nodes are tested to be outside of the view frustrum. Each of the child nodes which are at least partially visible is traversed. Then its eight child nodes are further tested for intersection with the view-frustrum. A branch of the recursion ends if a leaf node is reached or a node is completely outside or inside of the view-frustrum. All the visible leaf nodes which are containing grass tiles are further tested for occlusion as described in the next section.

Before computation it is important to enlarge each bounding box by the maximum extensions of the grass billboards. That is necessary as wind animation as well as collision response might have moved the billboards out of the bounding box. This situation especially occurs for billboards which are planted near to the boxes border.

6.3.1.2 Occlusion Queries

The axis aligned bounding boxes of the leaf nodes are used to test whether grass tiles are occluded. Therefore, the hardware based occlusion query [Sek04] is useful: An occlusion query returns the number of pixels of the render target influenced by the mesh's rasterized fragments [CG03]. Whenever no pixel is influenced the object is fully occluded, as is the case if all fragments failed the depth test. Therefore, as a first step the occluders, for example the terrain, are rendered to the depth buffer which as a result fills the depth buffer with the depth values of the occluders. For each bounding box that encloses grass billboards and has passed the viewport culling an occlusion query is performed. Again, each box is extended by the maximum size of a grass billboard. All grass tiles whose bounding box influences at least one pixel of the final image are still rendered.

6.3.2 Shading Grass Billboards

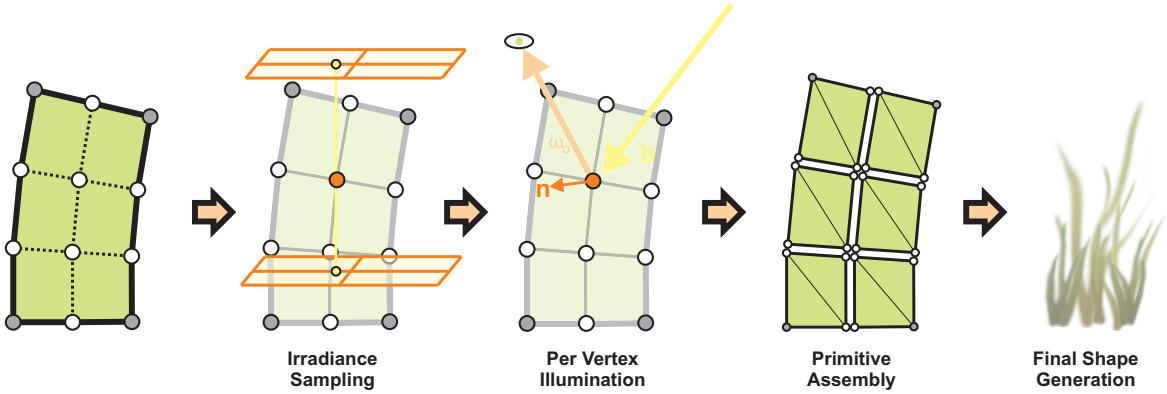


Figure 6.7: The billboards shading. The process is splitted into a per vertex illumination and a pixel based shape generation. The rasterized primitives are assembled during the rendering.

The shading process of the grass billboard uses a series of subsequent steps to generate the visual appearance of grass as shown in Figure 6.7. As a first step, the normal of each vertex is derived to allow for illumination computations: The irradiance at each vertex is combined with the reflection properties of the grass material. Then the billboard's primitives are assembled. Finally, the material color and the shape of the corresponding clump of grass are applied for each rasterized fragment. The shaded semi-transparent appearance is a result of the blending process based on multi-sampling.

6.3.2.1 Dynamic Irradiance Sampling

In case the collision system did not update the grass tile before the rendering, the wind animation described through Section 4.4 is applied to each of the undeformed billboards. Ultimately, during the dynamic sampling, the vertices of the billboard have to be up to date regardless of whether they are handled through the collision system or not.

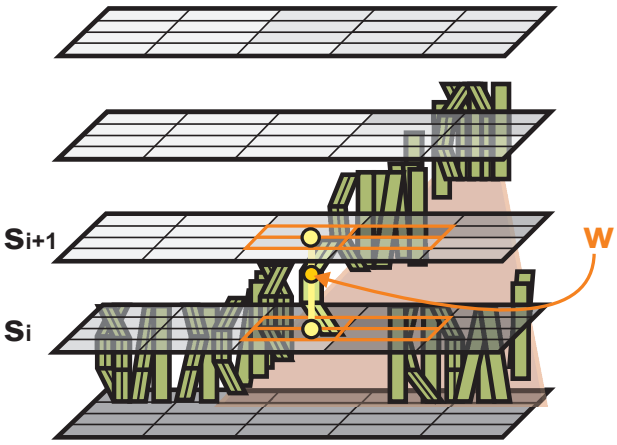


Figure 6.8: The dynamic sampling. The irradiance $\mathbf{E}$ as well as the bent normal $\mathbf{b}$ are interpolated for each vertex of the billboards within the irradiance volume. The two closest texture slices are interpolated bi-linearily. Afterwards, a shader based linear interpolation yields the final sample $\mathbf{w}$.

For each vertex $\mathbf{v}$ of the deformed or undeformed billboard the global illumination information is sampled within the pre-computed irradiance volume. A tri-linear combination of the eight closest voxels of the volume yields the desired information. As texture arrays are used, only bilinear interpolation among the slices is supported (see Section 3.1.2). That is the reason why the tri-linear interpolation is done in the shader. The information is interpolated between the samples of the closest texture slices as shown in Figure 6.8. Furthermore, the slices must be indexed directly. As the array is spanned along the y axis, the indices of the upper slice $s_{i+1} \in \mathbb{N}$ and the lower slice $s_i \in \mathbb{N}$ are:

$$\begin{aligned} s_i &= \quad \text{floor}(c_y \cdot n_{tex}) \quad \text{and} \\ s_{i+1} &= \quad \text{ceil}(c_y \cdot n_{tex}) \quad , \end{aligned}$$

where $n_{tex} \in \mathbb{N}$ is the number of texture slices of the volume. $\mathbf{c} = (c_x, c_y, c_z) \in [0,1]^3$ is the relative position of the vertex within the volume.[2] As the interpolation is done in the shader, the interpolation factor between both slices is defined as:

$$\alpha = n_{tex} \cdot c_y - s_i \ .$$

The implementation is shown in Code Sample 9.

Finally, the tri-linear interpolation within the volume turns out to be:

$$\mathbf{w} = (1-\alpha)\ \text{vol}(c_x, c_z, s_{i+1}) \ + \ \alpha\ \text{vol}(c_x, c_z, s_i)\ , \tag{6.9}$$

where $\mathbf{w}$ is the tri-linear sampled volume data. $\text{Vol}(x, y, s)$ is the bilinear sampling function among the slices. For each vertex both the irradiance $\mathbf{E}$ and the bent normal $\mathbf{b}$ are interpolated within the

[2] As the volume covers the whole grass layer the vertex of the billboard is always located within the volume.

```
inline
float4 GetRelativeCoord( in float3 Vtx )
{
    float4 c;
    // Compute the relative coordinate within the volume
    c = (Vtx - VolumeMinPos) * 1.0/VolumeSize;
    // Compute the index of the lower slice
    float LowerSlice = floor( NumSlices * c.y );
    // Compute the interpolation factor between both slices
    c.w = (c.y * NumSlices - LowerSlice);
    // Alter the relative coordinate to the index of the lower slice
    c.y = LowerSlice;

    return c;
}
```

Code Sample 9: The relative coordinate. The values **VolumeMinPos** and **VolumeSize** are passed to the shader. They define the bounding volume of the grass layer. Please note that only the index of the lower slice is computed. The adjacent slice is found at the following index.

volume (see Section 6.2). As the interpolation of unit length vectors does not necessarily yield a unit length vector the interpolated bent normal is normalized after interpolation. The implementation of the irradiance information is shown in Code Sample 10. As a result the sampling allows to dynamically obtain global illumination information per vertex.

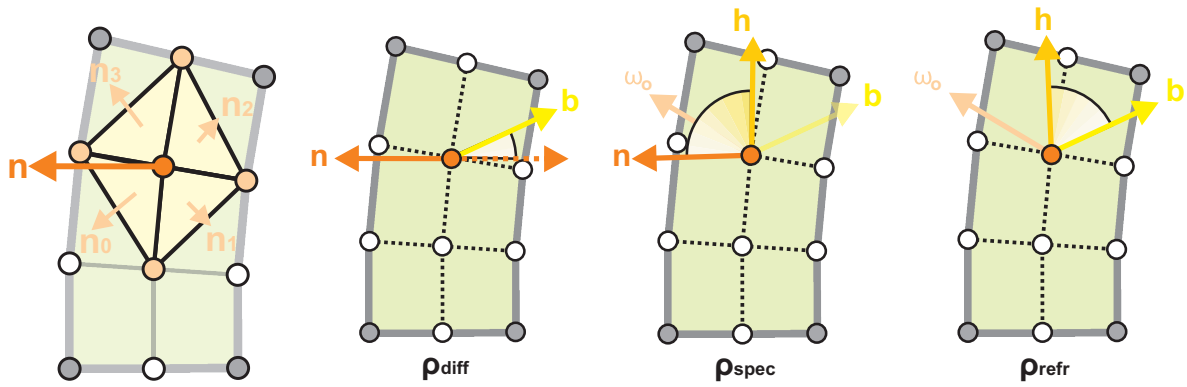
```
inline
float3 GetIrradiance( in float4 c )
{
    float3 irradiance[2];
    // receive the bi-linearly interpolated irradiance value of both slices
    irradiance[0] = IrradianceArray.SampleLevel( LinearMirrorSam, c.xzy, 0 );
    c.z += 1.0;
    irradiance[1] = IrradianceArray.SampleLevel( LinearMirrorSam, c.xzy, 0 );
    // return the value received by tri-linear interpolation
    // between the two closest slices
    return lerp( irradiance[0], irradiance[1], c.w );
}
```

Code Sample 10: The irradiance sampling. The relative coordinate inside the irradiance volume is used to sample the irradiance information. The closest slices of the texture array are addressed by the y-coordinate. For the sampling of the ambient occlusion information another function is called.

As the irradiance $\mathbf{E}$ approximates all the incoming light relating to the amount of occlusion and with respect to the cone of incoming light the last term which is missing in Equation 6.2 is the reflectance term $\rho(\mathbf{p}, \omega_i, \omega_o)$. The illumination computation is based upon this information.

6.3.2.2 Per-Vertex Illumination

Due to the complex illumination computations and the massive amount of grass billboards per pixel lighting is avoided. Instead of this, Gouraud shading is implemented which shifts the work to the geometry shader. Each normal $\mathbf{n}$ is set up with regard to all adjacent facets of the vertex v. All the



(a) (b) (c) (d)

Figure 6.9: The per vertex illumination. In Figure 6.9(a) the assembly of the vertex normals is shown. Each vertex normal is setup with regard to the adjacent triangles of the vertex. From Figure 6.9(b) up to Figure 6.9(d) the components of the billboard's reflectance term are illustrated.

normals $\mathbf{n}_t$ of the triangle primitives that contribute to the normal are integrated as can be seen in Figure 6.9(a):

$$\mathbf{n} = \frac{\sum_{t=0}^{k} \mathbf{n}_t}{\left\| \sum_{t=0}^{k} \mathbf{n}_t \right\|} , \tag{6.10}$$

where k is the number of the triangles that surround the vertex v. The resulting normals lead to a smooth Gouraud shading all over the surface.

Within this step, the reflectance term can be computed as the irradiance $\mathbf{E}$ and the bent normal $\mathbf{b}$ are obtained for each vertex $\mathbf{v}$. The term constitutes the local reflection properties of grass and is split into several components as presented in Figure 6.9:

$$\rho = \rho_{diff} + \rho_{spec} + \rho_{refr} . \tag{6.11}$$

The first component is the diffuse term which is affected by the normal $\mathbf{n}$ and the sampled average incoming light direction $\mathbf{b}$ (see Figure 6.9(b)):

$$\rho_{diff} = (|\mathbf{n} \bullet \mathbf{b}| \cdot 0.7 + 0.3) \cdot (1, 1, 1) ,$$

where $\rho_{diff} \in [0.3, 1]^3$ is the amount of diffuse reflection for all channels RGB. It is constituted by the angle between the incoming light direction and the normal of the vertex $\mathbf{v}$. The more the directions differ, the smaller the value of the diffuse term. If both directions are perpendicular respectively the surfaces faces away from the light the contribution is minimal. As the billboards are flat the backfaces are visible as well. Therefore, the absolute value of the diffuse term is computed in order not to darken the back faces by mistake. The quad of the billboard should also not turn to black if it is being viewed

from the side as this causes an unnatural appearance. Therefore, the range of the diffuse term $[0, 1]$ is mapped to the range $[0.3, 1]$ by an additional ambient amount.

The specular term adds highlights in respect to the view direction (see Figure 6.9(c)):

$$\rho_{spec} = (1 - f)\ \max(\mathbf{n} \bullet \mathbf{h}, 0)^4\ \cdot\ (1, 1, 1)\ ,$$

where $\rho_{spec} \in [0, 1]^3$ is the amount of specular reflection for all channels RGB. $\mathbf{h} = \frac{\omega_o + \mathbf{b}}{\|\omega_o + \mathbf{b}\|}$ is the normalized halfway vector between the view direction ω_o and the average light direction $\mathbf{b}$. The effect of this so-called Blinn-Phong lighting is that brighter highlights are added where the view direction and the light direction are closely aligned. The maximum is reached if both vectors coincide. Because grass tends to have great highlights over its surface the exponent is kept small. The interpolation factor $f \in [0, 0.5]$ which accounts for the semi-transparent nature of grass blades signifies the percentage of refracted light described in short: The more refracted light affects the billboard, the more reduced is the contribution of the specular component and the higher is the value of f. In cases where the grass billboards are lit from behind and viewed from ahead the refracted light contributes to illumination. Therefore the term f is determined relating to the angle between the halfway vector and the average incoming light direction $\mathbf{b}$:

$$f = \max(\mathbf{h} \bullet -\mathbf{b}, 0) \cdot 0.5\ .$$

The max() function prevents the term from falling below zero and the factor reduces the influence of back face lighting. This term is similar to the Fresnel Term [MH99].

The more the refracted light affects the illumination the more the material color turns to yellow as is for plants modeled through a color shift $\mathbf{s}_{refr} = (1.0, 0.9, 0.3)$ [KCS07]:

$$\rho_{refr} = f\ \cdot\ \mathbf{s}_{refr}\ ,$$

where $\rho_{refr} \in [0, 0.5]^3$ is the refracted light.

Combining all components as a result yields the reflectance term $\rho \in \mathbb{R}^3$. ρ is a variation of the so-called Blinn-Phong model common to computer graphic applications [MH99]. As a result of combining both, the global irradiance and the local reflectance properties, a coarse approximation to function 6.2 is computed. However, the final color is evaluated per pixel. The reason for the divided process is that the surface material $\mathbf{c}_{mat} \in [0, 1]^3$ is stored in the semi-transparent decal texture which is necessary for the final shape generation. So, the reflectance term is multiplied with the irradiance in order to interpolate as few parameters as possible throughout the rasterizer stage:

$$\mathbf{e}_{refl} = \rho\ \otimes\ \mathbf{e}\ .$$

The implementation is shown in Code Sample 11. After all illumination computations each of the quad primitives are assembled as described next.

```
inline
float3 Illuminate( in float3 Vtx, in float3 Norm )
{
    // Compute the relative coordinates.
    float4 c = GetRelativeCoord( Vtx );

    // Sample the bend normal and the irradiance
    // for the vertex
    float3 b = GetBendNormal( c );
    float3 E = GetIrradiance( c );
    float3 viewDir = EyePos - Vtx;
    float3 h = normalize( viewDir + b );
    // How much of the light is transmitted?
    float f = saturate( dot(h,-b) ) * 0.5;

    // Evaluate the diffuse term
    float diff = abs(dot(Norm,b)) * (1-AMBIENT_AMOUNT) + AMBIENT_AMOUNT;
    float3 diffColor = diff;

    // Evaluate the specular term
    float spec = saturate( dot(Norm,h) );
    spec *= spec;
    spec *= spec;
    float3 specColor = (1-f) * spec;

    // Evaluate the color shift which simulates transmitted light
    float3 refrColor = f*ColorShift;

    // Return the reflected irradiance
    return E * (diffColor + specColor + refrColor);
}
```

Code Sample 11: The illumination. The illumination function is called for each vertex which is part of an assembled quad primitive. The values **ColorShift** and **EyePos** are constant for each frame.

6.3.2.3 Primitive Assembly

As the grass billboards are stored in a single point list the final triangle primitives are created during the geometry shader. Either the deformed or the undeformed representation of the billboard exists depending on whether recovering time is left or not. For each billboard up to six quadrilaterals are created depending on its deformation state. When the undeformed state is rendered a single quad $\mathbf{Q}(\mathbf{v}_{0,0}, \mathbf{v}_{3,0}, \mathbf{v}_{0,2}, \mathbf{v}_{3,2})$ is streamed. Six quads are streamed in case that recover time is left:

$$\mathbf{Q}(\mathbf{v}_{v,h}, \mathbf{v}_{v+1,h}, \mathbf{v}_{v,h+1}, \mathbf{v}_{v+1,h+1}) \;\; \forall\, v = \{0,1,2\} \wedge \;\; h = \{0,1\} \;\; .$$

The primitive assembly is shown in Code Sample 12.

After that, each of the vertices receives a texture coordinate. This coordinate is used for projecting the decal image onto the assembled primitives respectively rasterized fragments. As the whole image is addressed through the texture coordinate range $[0,1]^2$, the coordinates are equidistantly distributed over the billboard's vertices $\mathbf{v}_{i,j}$:

$$\mathbf{t}_{i,j} = (\frac{i}{3},\; \frac{j}{2},\; id_{image}) \in \mathbb{R}^3 \;\; ,$$

where id_{image} is the procedural assigned index into the texture array containing the semi-transparent images (see Section 4.3.2.3 and Section 4.1.3). Streaming the previously computed $\mathbf{e}_{refl}$ value and the texture coordinates per vertex allows to compute the final appearance on a per pixel level.

```
inline
void StreamDeformedBillboardQuads(
    inout  TriangleStream<BILLBOARD_VTX> BillVtxStream,
    in     float3                        Vtx[VTX_CNT],
    in     float                         ImageId )
{
    BILLBOARD_VTX BillVtx[4];
    // 1.)... Compute the texture array coordinates
    //        into the decal image (imageId is used)
    // 2.)... Compute the illumination for each vertex
    // 3.) Assemble and stream each quad of the deformed
    //     billboard
    int Idx[4];
    for( int v = 0; v<(VTX_CNT_VERTICAL-1); v+=1 )
    {
        for( int h = 0; h<(VTX_CNT_HORIZONTAL-1); h+=1 )
        {   // Select the current vertices
            Idx[0] = v          *(VTX_CNT_HORIZONTAL) + h;
            Idx[1] = (v+1)      *(VTX_CNT_HORIZONTAL) + h;
            Idx[2] = v          *(VTX_CNT_HORIZONTAL) + (h+1);
            Idx[3] = (v+1)      *(VTX_CNT_HORIZONTAL) + (h+1);

            // Compute the projection space coordinates.
            for( int q=0; q<4; ++q )
                billVtx[q].ndcPos = mul( float4(Vtx[Idx[q]], 1),
                                         ViewProjection );

            // Stream the quad
            BillVtxStream.Append( BillVtx[0] );
            BillVtxStream.Append( BillVtx[1] );
            BillVtxStream.Append( BillVtx[2] );
            BillVtxStream.Append( BillVtx[3] );
            BillVtxStream.RestartStrip();
        }
    }
}
```

Code Sample 12: The primitive assembly. The illumination is computed and the texture coordinates are assigned for each vertex. Afterwards, the function assembles the quad primitives of the deformed billboards. These quads are passed through the rasterizer as a series of triangle strips. In case the undeformed billboard is rendered, another function is called which only streams the quad made up by the edge vertices.

6.3.2.4 The Final Shape

Even then all the work done so far is based on vertices the final shape of the grass as well as the application of the illumination is done on the base of each pixel: As each clump of grass blades is represented by a semi-transparent decal texture, as mentioned in Section 4.1.3, now for each pixel of the quadrilateral primitives the decal texture is sampled. The texture coordinates and the reflected irradiance are computed for each vertex throughout the primitive assembly described in the previous

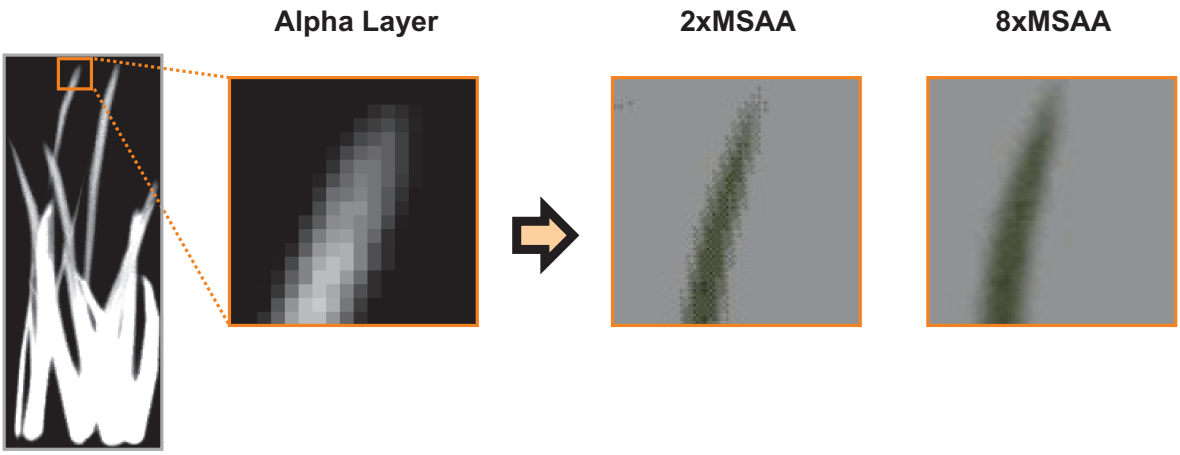


Figure 6.10: The edge smoothing. The edges of the grass billboards are smoothed by a slight transition between opaque alpha values and transparent alpha values. Please note that the multi sampling resolution determines the final image quality (compare the result of 2xMSAA with the result achieved by 8xMSAA).

section. As a result of the sampling the material color $\mathbf{c}_{mat} \in [0,1]^3$ as well as the transparency $a_{mat} \in [0,1]$ are obtained. The final color $\mathbf{c}_{final}$ for each pixel is computed by multiplying the material color with the reflected irradiance:

$$\mathbf{c}_{final} = \mathbf{c}_{mat} \otimes \mathbf{e}_{refl}$$

The pixel shader is outlined in Code Sample 13.

```
float4 FinalShape( BILLBOARD_VTX Pix ) : SV_TARGET
{
    // Sample the decal texture
    float4 DecalColor = DecalTextures.Sample( AnisotropWrapSam, Pix.Texcoord );
    // Combine the grass color with the reflected irradiance
    DecalColor.xyz *= Pix.ERefl;

    return DecalColor;
}
```

Code Sample 13: The final shape. The sampled color for each pixel is modulated with the reflected irradiance. This results in the final color for the pixel. The alpha value of the sampled color is returned in order to allow for blending.

The thin surface of grass blades does also provide the ability to look through them. That is noticed best if the viewer is near the grass blade. This semi-transparent nature of the grass is simulated with the aid of the alpha-to-coverage feature (see Section 3.1.2). The blending between billboards is done without the necessity of performing expensive depth-sorting whenever the viewer's position changes. Sub pixels are filled by grass billboards with respect to the transparency value a_{mat}. The higher the value of the grass billboard's pixel the more sub pixels of the render target are filled with the pixel's

color. The blending occurs during downsampling of the multisample resolution to the final image resolution [Mye06]: All subpixels are summed up to the final pixel color. As a restriction only the most recently rendered grass billboards affect the sub pixels which determine the final image. But due to the chaotic nature of grass this is not easy to notice and thus, does not have an effect on the visual results.

The shape of the final grass is also set up by the alpha layer of the decal image. At every pixel of the quad where the sampled transparency is zero no pixel is rendered. Furthermore, the edges of the grass billboards are smoothed. This is a consequence of the slight transition between transparent areas where no blades are placed and opaque areas where blades define the shape of the clump of grass as shown in Figure 6.10.

Chapter 7

Results

This chapter presents the results of GPU-Based Responsive Grass. After outlining the visual quality of the collision system and the rendering system, the performance of both is discussed. Furthermore, the application to modern real-time engines is expressed.

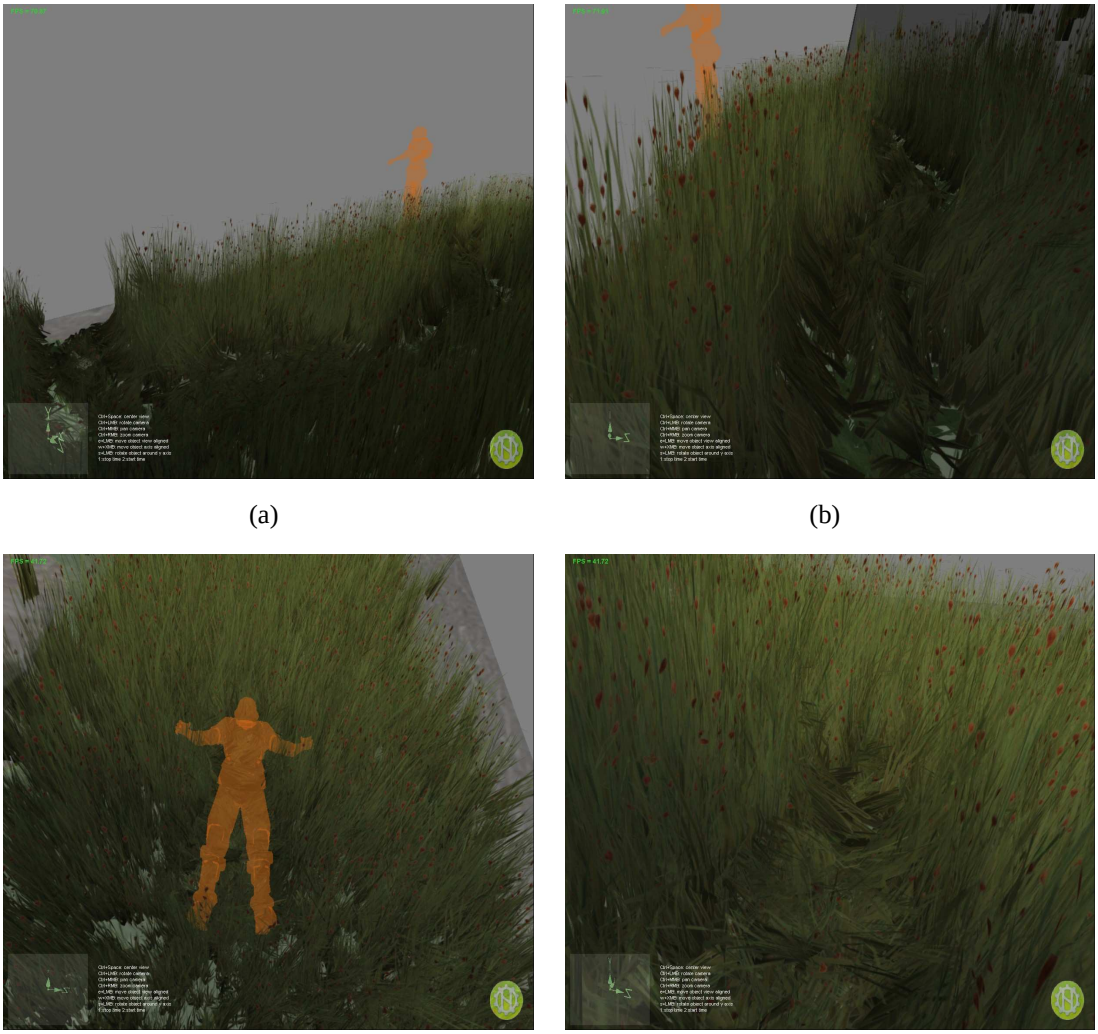
7.1 Visual Quality

Throughout this section the visual results of the collision handling and the rendering process are described. Example images of different scenes are used to illustrate several components of this system. In addition, some unpleasant side-effects of the techniques are pointed out.

7.1.1 The Collision Handling

The collision response has the most important influence on the perception of the responsive grass layer. Therefore this section shows some example images concerning the visual results of the collision response and recovering.

The first images Figure 7.1(a) and Figure 7.1(b) show the response of the grass after the scene object has moved through the meadow. The scene is rendered with 60 frames per second by using fourfold multi sampling anti aliasing (4xMSAA). The grass layer contains 46000 grass billboards. The whole grass on the line of movement is pushed to the side or stamped down (Note Figure 7.1(a)). The space between the legs of the collision mesh has left a trail in the tramped grass as can be seen in Figure 7.1(b). Figure 7.1(c) shows the scene object which is resting in the middle of the meadow. As can be seen by observing Figure 7.1(d) the object has left a clearly noticeable imprint on the grass. The collision mesh's shape is easy to estimate within the grass. The results of the recovering process are presented in Figure 7.2: The scene object is moved along a straight line (of motion) in the dense meadow. After the collision has taken place, until the shape is not fully recovered the flattened grass billboards smoothly rise back to their original form. The response and the recovering yield nice results in case of chaotic and dense meadows.



(a) (b)

(c) (d)

Figure 7.1: The collision response in a dense meadow. The images show the result of the collision handling in a dense field of grass. In Figure 7.1(a) and Figure 7.1(b) the collision mesh has left a clearly visible track in the grass. In Figure 7.1(c) and Figure 7.1(d) the imprint of the scene object is shown.

The collision response provoked in areas of sparsely planted grass billboards is displayed in Figure 7.3. The images show the scene object which is slowly grabbing something in the grass. The arm which collides first produces an undesired response: Instead of pushing the grass in the direction of the movement the grass is being pushed to the sides. That situation occurs because the arm of the collision mesh has not a sphere-like but rather a longish shape. For non sphere-like shapes, sometimes the wrong reaction direction is looked up in the depth cube. This results in an unexpected response. The scene object has to be approximated by a series of coarser sphere-like meshes in order to avoid such unnaturally reactions. In addition, the angular appearance of deformed grass billboards is clearly

(a) (b) (c)

Figure 7.2: The recovering of tramped grass. In this figure the process of the recovering is shown. The scene object has left a clearly visible line of movement in the meadow. After a while the grass smoothly recovers. Note that the grass is darkening when it is tramped down.

(a) (b) (c)

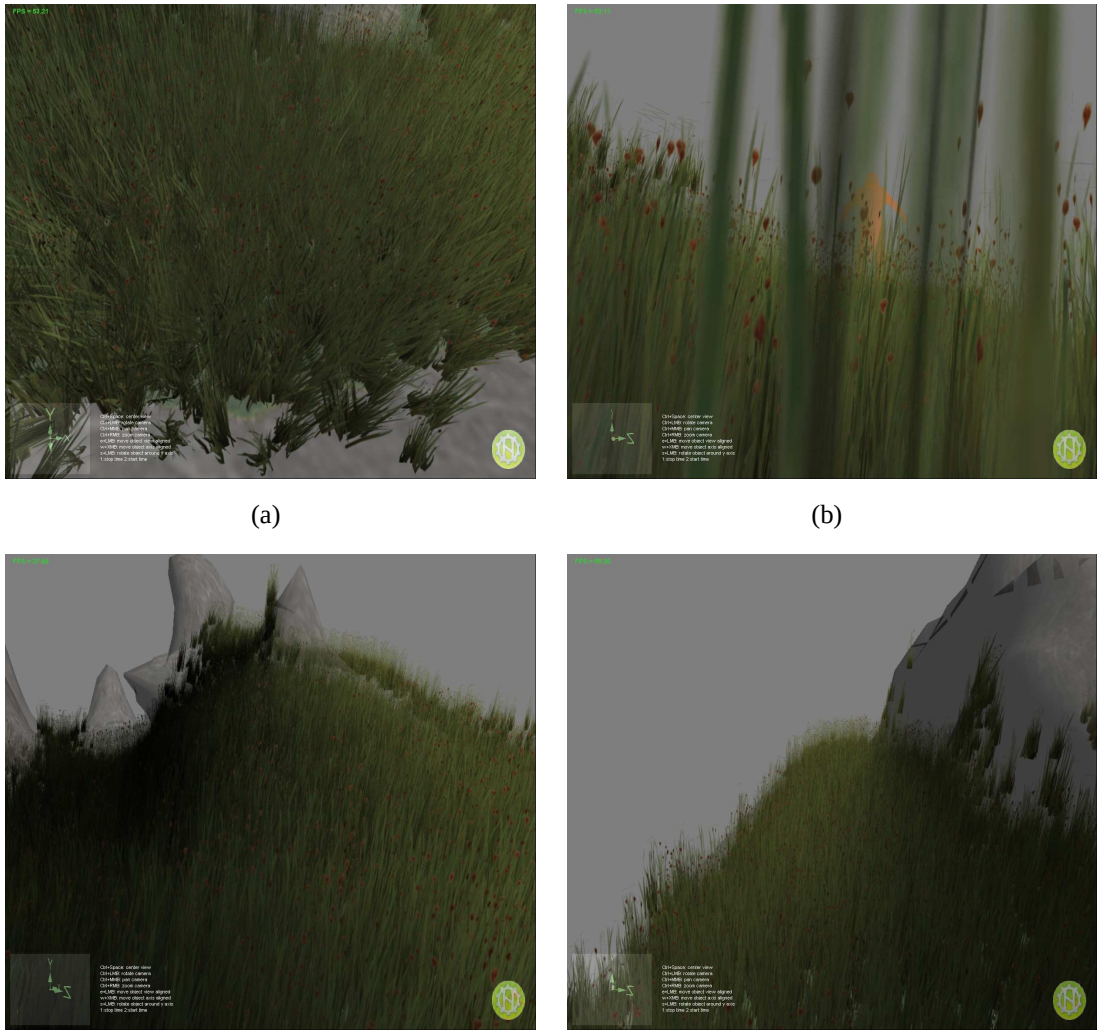
Figure 7.3: The squared look. The figures show a sparsely planted region. From the left Figure 7.3(a) to the right Figure 7.3(c) the object is moved toward the grass clumps on the right (Please note that the object is in a unnatural way beyond the terrain in order to make the movement possible). The longish shape of the arm causes an unnatural reaction: The billboards are pushed to the side of the arm instead of being pushed in the direction of the movement. In addition, the squared look of deformed billboards is easy to estimate.

visible. However, these unpleasant reactions are only noticeable in those regions of the grass layer where grass billboards are sparsely planted.

7.1.2 The Rendering System

The visual appearance of the grass billboards is very important. This aspect is determined by closely planted grass and realistic illumination. In this section the results of each component of the rendering process are shown.

In Figure 7.4(a) all components of the reflectance term are visible at the same time. On the left



(a) (b)

(c) (d)

Figure 7.4: The visual results of the rendering process. In Figure 7.4(a) the grass is shown from the top view. The results of the reflectance term are visible: from the left to the right the specular term is replaced by the refraction term. In Figure 7.4(b) the scene object shimmers through the semi-transparent grass billboard. In Figure 7.4(c) and in Figure 7.4(d) the outcome of the global illumination applied to the grass layer is visible.

the specular term is visible. Please note the bright highlights in contrast to the grass shown in the middle of the figure. On the right side the effect of the transmitted light is shown. The grass is slightly shifted to a more yellowish color. The semi-transparent nature of the grass billboards is visible in Figure 7.4(b). The viewer is close to a grass blade and the scene object shimmers through it as a result of the Alpha-to-Coverage feature.

In both, Figure 7.4(c) and Figure 7.4(d), the global illumination becomes visible. The random directions used to create the occlusion information for the grass layer are mainly concentrated along

a single direction. The smooth transition between shadowed and lit areas of the grass layer creates an idea of the overall illumination of the scene. In the left half of Figure 7.4(c) the grass is darkened by a hill which is located in front of it. From the left to the right side the grass is more and more illuminated. This makes it easy to identify the light direction. In Figure 7.4(d) the light is directed to the viewer. The rock in front of the viewer casts smooth shadows to the right half of the grass area.

The dynamic illumination is visible if the grass is pushed down as can be seen by comparing the images shown in Figure 7.2. Whenever the scene object moves through the grass, the grass tramped down is darker as can be seen in Figure 7.2(a). That is because the dynamic sampling described in Section 6.3.2.1 interpolates values between the closest slices of the irradiance volume. The more the grass is tramped down, the more raise the influence of the lower slice. The lowest slice in general is occluded by the terrain. Hence, a grass billboard receives less irradiance if it is tramped down. That simulates the shadow which is caused by neighboring blades. The more the grass recovers the less it is affected by the lowest slice of the volume. However, this effect relates only to those billboards which are located on the lowest level of the grass layer.

7.2 Performance Analysis

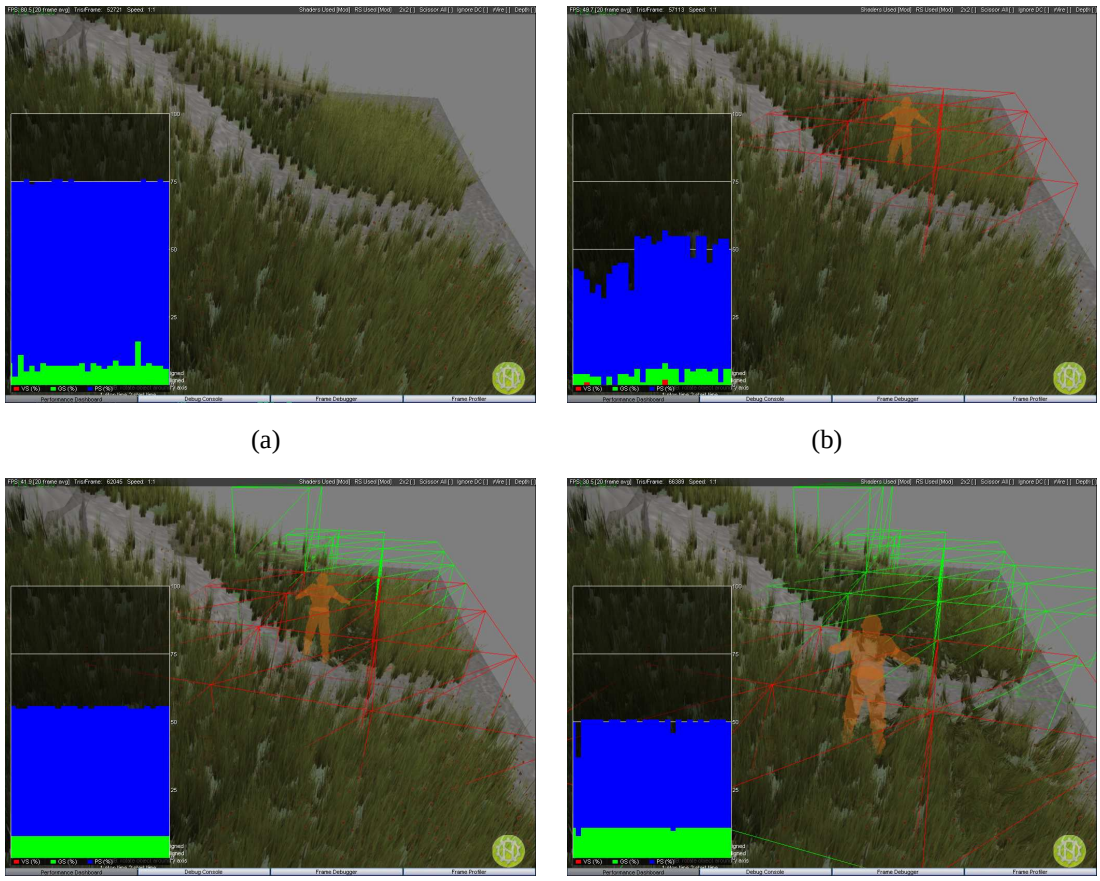
Achieving a high performance is one of the major aims of real-time applications. All parts concerning the grass layer are designed to reduce the workload of the CPU as much as possible. Therefore the handling of the billboards is almost completely shifted to the GPU. That is why in this section the performance of responsive grass is analyzed with aid of the NVidia PerfHUD[1] tool. This tool helps to identify performance bottlenecks on the GPU. All the tests are made on a AMD Athlon 64 3500+ 2.2 GHz processor and a GeForce 8800 GTX graphics card with 768 MB DDR3 memory.

7.2.1 Collision Handling

The performance analysis of the collision handling is done by observing the same scene with different collision conditions. Only a single collision object is used during the tests. Figure 7.5 shows the four constellations of the same scene running at 30-80 frames per second. Figure 7.6 shows how much time is consumed in each GPU pass (of either the collision pipeline or the rendering process). The scene contains approximately 37000 billboards which requires 12 MByte of graphics memory. Each time the image is rendered with Alpha-to-Coverage enabled and 4xMSAA. Both culling techniques prune away all grass tiles that can not be seen at all.

In Figure 7.5(a) the grass layer which smoothly waves in the wind is rendered with 80 frames per second. Referring to the diagram 7.6(a) the only pass that causes computations, is the rendering pass of the unaffected grass tiles (RN). The scene is pixel bound as can be noticed by observing the utilization graph in Figure 7.5(a). The graph shows the workload balancing of the programmable stages of the

[1]The NVIDIA performance monitor tool is a copyright of the Nvidia Cooperation and comes for free with the restriction of usage only for the NVidia products. The tool makes it possible to analyze the video cards performance at real-time.

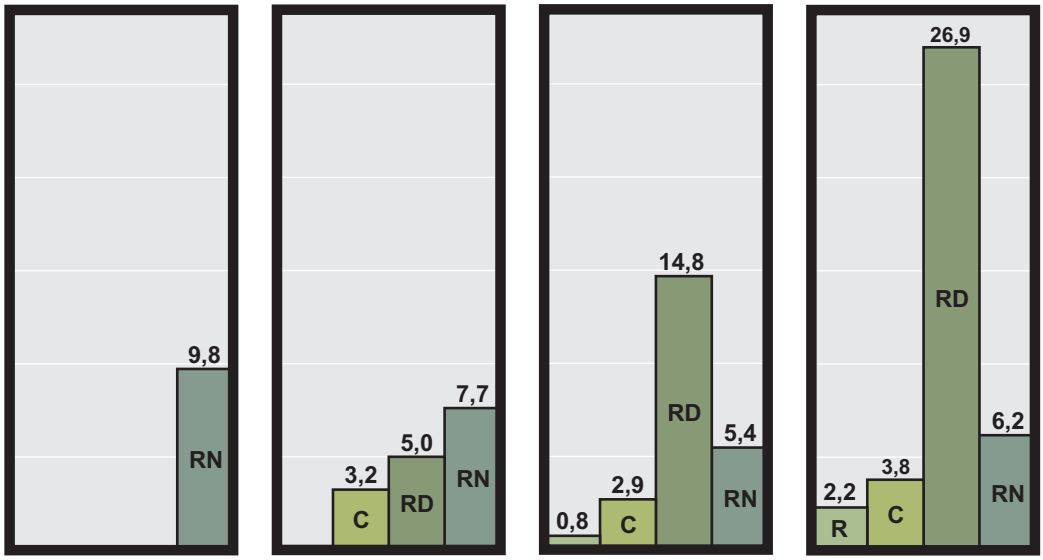


(a) (b)

(c) (d)

Figure 7.5: The collision conditions. The figure shows four images with a increasing number of deformed grass billboards. In addition, the colliding (green) and recovering (red) leaf nodes of the octree are displayed in the scene. In Figure 7.5(a) no deformation has occurred at all. From Figure 7.5(b) to Figure 7.5(d) the number of deformed billboards increases. The graph on the left corner in all the figures displays the shader utilization. Each bar shows the percentage of workload caused by the corresponding shader unit during the last frames: The blue bar shows the utilization of the pixel shader unit and the green bar shows the utilization of the geometry shader unit. Accordingly a red bar determines the workload caused by the vertex shader.

rendering pipeline: The unified streaming processors are utilized to work on pixels with about 75 per cent (the blue bar) whereas the geometry shader unit of the pipeline is active by approximately ten per cent (the green bar). As the vertex shader (red colored) only passes the points to the geometry shader it has no influence on the performance at all. Hence, the vertex shader does not even demand the streaming processors. The remaining workload is caused by frame buffer operations. In all scenes approximately 16 million pixels are processed within the fragment shader resulting in many read as well as write accesses to the frame buffer. These are amplified by the Alpha-to-Coverage feature which requires a multisample resolution that in this case is four times higher as the image resolution.



(a) (b) (c) (d)

Figure 7.6: The performance of the collision handling. The series of diagrams show the time consumed throughout the different scenes of Figure 7.5. Figure 7.6(a) up to Figure 7.6(d) show the performance consumed throughout the scenes shown from Figure 7.5(a) up to Figure 7.5(d). For each GPU pass a performance bar is shown. These are the update of the recovering grass tiles (R), the update of the colliding grass tiles (C), the rendering of the affected grass tiles (RD), and the rendering of the unaffected grass tiles (RN). Each duration is given in milli-seconds.

In the second constellation shown in Figure 7.5(b) the grass layer is affected by the collision object. The nine grass tiles respectively leaf nodes that are tested through the collision pass are displayed by red wireframe boxes. Hardly any of the grass billboards are affected except those that are near the collision object. The utilization graph in Figure 7.5(b) shows that more workload is caused by other parts of the pipeline as the percentage of the pixel shader is reduced in contrast to the previous case (note the 20 per cent decrease of the blue bar compared with Figure 7.5(a)). This time is consumed in the stream output stage which is the back-end of the collision pass. As can be noticed by looking at the diagramm 7.6(b) three passes are executed on the GPU. The nine grass tiles are processed by the collision pass (C) which takes the least of all computational time. In the second pass (RD) those tiles are rendered. The rendering of the rest of the grass layer (RN) is consuming the most time as it envolves most of the grass billboards. The scene is rendered at 50 frames per second.

In Figure 7.5(c) more leaf nodes of the octree are affected. The green boxes show the recovering grass tiles which are updated by the recover pass. In addition, more grass billboards are deformed. Consequently, the performance of the render pass which covers the updated grass tiles (RD) is decreased. As shown in diagram 7.6(c) the consumed time for rendering those affected billboards is much higher than the time which is necessary to render the undeformed ones (RN). This time overhead is caused by the primitive generation in the geometry shader as well as by the rendering of the

high count of primitives. The collision pass (C) in this case stays nearly constant because only one single collision object effects the grass layer. The number of collision tests does not increase. The overall performance is still pixel bound.

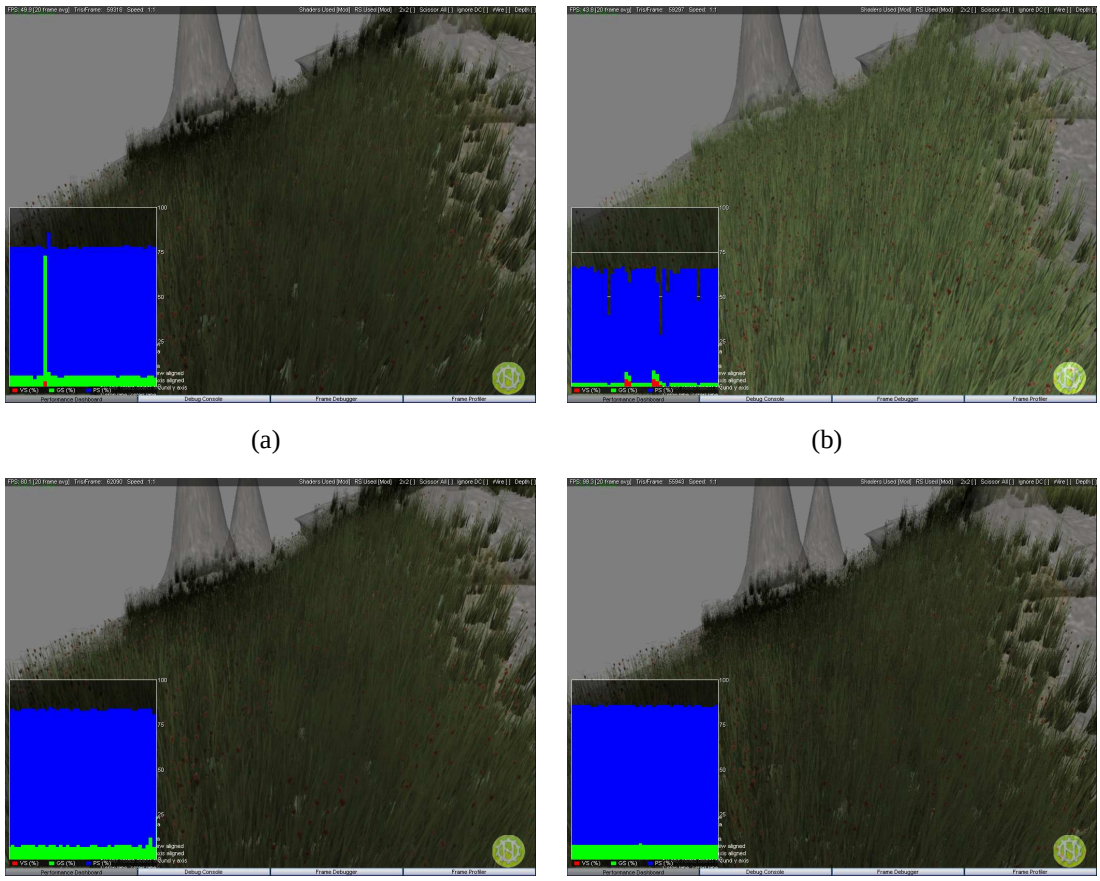
The last constellation which is shown in Figure 7.5(d) comes up with nearly the same number of grass tiles that are updated during the collision handling. But almost every billboard which is part of the recovering tiles (the green boxes) is deformed. As a result there is a performance loss because the geometry shader has to assemble much more quad primitives (Note the increased percentage of the green bar of the utilization graph in respect to the prior cases). In addition, the deformed billboards cause more workload on the geometry shader as more primitives are created. This can be discovered by observing the high duration of the render pass (RD) of the collided and recovering grass tiles in the diagram 7.6(d). Furthermore, the primitives are cached on graphics memory which leads to many read and write operations. In such cases where a great amount of billboards is deformed the frame rate decreases to 30 frames per second.

To summarize, the performance of collision handling depends on the number of primitives that are generated and passed through the rasterizer back-end. Both, the memory operations as well as the work that has to be done in the geometry shader stage, are affected during the rendering. Consequently it is necessary to setup a low recovering time in order to preserve the overall performance. The computational time which is necessary for the collision handling remains constant (compare the diagrams in Figure 7.6) and depends on the number of scene objects. However, the performance is still bound by the rasterizer back-end and the frame buffer accesses as referred to in the following section.

7.2.2 Rendering Process

The rendering system is designed to obtain good global illumination at minimal computational costs. Therefore the global illumination is done in a preprocessing step. During runtime the computation is reduced to the dynamic sampling described in Section 6.3.2.1. In this section it should be shown that the illumination does not have significant influence on the overall performance. Furthermore, the performance bottleneck is pointed out. Therefore, a scene containing more than 46000 grass billboards is regarded with different settings.

In Figure 7.7(a) this scene is rendered with global illumination and 8xMSAA. 50 frames per second are rendered. The raster operations as well as the frame buffer accesses are as busy as the shader units. This can be noticed by observing the utilization diagram 7.8(a) which shows how busy each unit of the rasterizer back-end was during a single draw call. In addition, the total frame time (FT) is displayed (50 frames per second). In cases where global illumination and 8xMSAA are enabled, the billboards cause a great amount of workload in the rasterizer back-end. The streaming processors (which in total require the time of the USH bar in the diagrams of Figure 7.8) almost completely work on the pixel level as displayed by the blue bar of the utilization graph in Figure 7.7(a). Consequently, it is absolutely necessary to implement a per vertex illumination due to the high amount of pixel work-



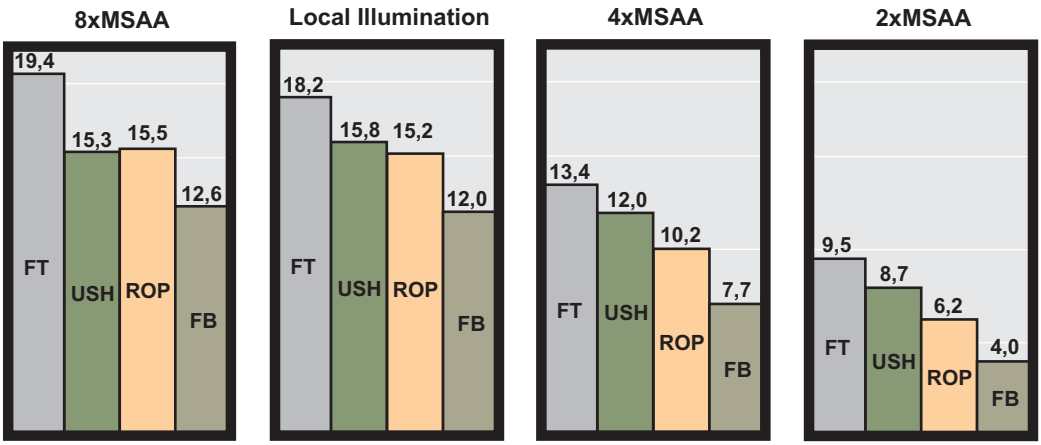
(a) (b)

(c) (d)

Figure 7.7: The results of different settings. In Figure 7.7(a) the scene is rendered with global illumination and 8xMSAA antialiasing. In Figure 7.7(b) the illumination model is replaced by a simple diffuse term. In Figure 7.7(c) the multisample resolution is reduced to 4xMSAA and in Figure 7.7(d) the resolution is further reduced to 2xMSAA. The utilization of the programmable units is shown in the lower left of the figures. The blue bar represents the utilization of the pixel shader. The green bar shows the utilization of the geometry shader.

load. In Figure 7.7(b) the illumination model is reduced to a simple diffuse term which replaces the global illumination model described in Section 6.3.2.2. This has no affect on the overall performance as can be made out by comparing diagram 7.8(a) with diagram 7.8(b). The diagram 7.8(b) displays the utilization of the units while the diffuse term is used. The span of time consumed in the shader unit (USH) as well as the time which is spent on the per pixel operations (ROP + FB) remains almost unchanged. Since illumination is implemented on a per vertex level and the main time is consumed in the rasterizer back-end the performance (see the frame time) is not influenced.

The pixel shader code of the render process is kept as short as possible (see Section 6.3.2.4). As the pixel operations are utilized with almost the same amount as the pixel shader, the multisample resolution of the Alpha-to-Coverage feature is reduced next: In Figure 7.7(c) the scene is rendered with



(a) (b) (c) (d)

Figure 7.8: The utilization of the rendering pipeline units. The diagrams show the time consumed by the most utilized units during the rendering of Figure 7.8(a) (8xMSAA), Figure 7.8(b)(Local Illumination), Figure 7.8(c) (4xMSAA) and Figure 7.8(d) (2xMSAA). In each diagram the utilization of the unified shader (USH), the raster operations (ROP) (for example the depth test), the frame buffer accesses (FB) and the overall frame time (FT) are displayed in milli-seconds. As can be noticed, the frame time significantly depends on the multisample resolution instead of which the illumination model does not have an effect on the overall performance (compare Figure 7.8(a) with Figure 7.8(b)).

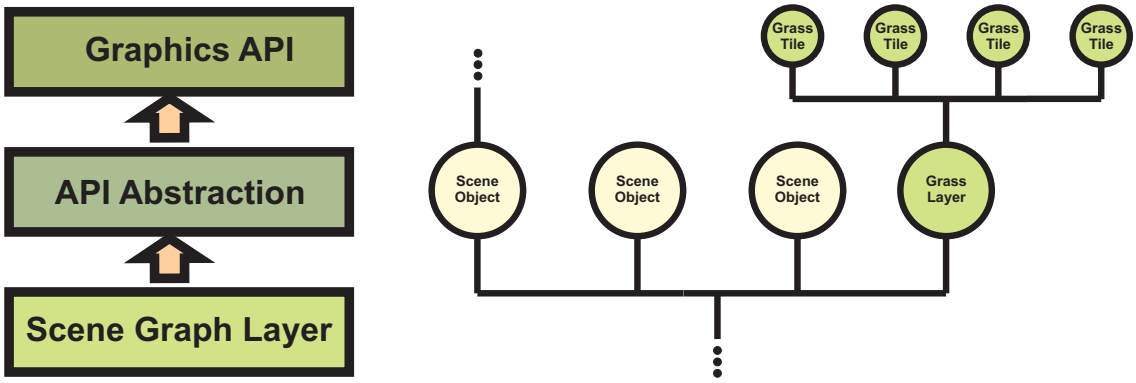
4xMSAA which results in an increased frame rate (FT) (80 frames per second). Both the rasterizer operations (ROP) and the frame buffer accesses (FB) are reduced as can be seen in diagram 7.8(c). In addition, the multisample resolution is further reduced which results in an additional performance enhancement of 20 frames (Note the diagram 7.8(d)).

As already shown throughout the diagrams in Figure 7.8 the workload significantly depends on the multisample resolution whenever a great amount of billboards is rendered. With respect to the multisample resolution, the performance can be significantly improved. However, the multisampling determines the visual quality of the blending process as already described in Section 6.3.2.4.

7.3 Embedding

Most real-time applications employ scene graph engines in order to manage the complexity of large scenes. The grass layer is embedded in such an engine to improve the usability of the presented techniques. A DirectX 10 version of the Nebula 2 engine therefore has been extended.[2] The engine consists of different layers each of which builds an abstraction to its subjacent level. This is a common way to build up a manageable structure on the top of the graphics API as illustrated in Figure 7.9(a). All components of the scene graph are modularly built. Objects respectively entities are managed by

[2]The nebula engine is a copyright of Radon Labs GmbH.



(a) (b)

Figure 7.9: The scene graph layout. In Figure 7.9(a) the common layout of a engine is shown. The scene graph structure into which the grass layer is embedded is presented in Figure 7.9(b).

a scene graph hierarchy. Each of those entities is represented by a scene graph node.

As illustrated in Figure 7.9(b) the grass layer is implemented as a scene graph node as well. This node comprises all parts which are required by the grass system. Using a scene graph enables to dynamically extend or replace components of the system. The leaf nodes of the octree are implemented in form of independent child nodes of the grass layer. These child nodes are also part of the scene graph. This structure allows the leaf nodes to be looked up within the scene graph. In addition, scene objects that are able to influence the grass layer are easily referenced throughout the scene graph tree as well. All the interfaces are well-defined. This makes it possible to offer a higher abstraction of the supported functionality. The modularity of the components is further increased due to the distributed implementation. The CPU makes predecisions on the base of the octree nodes and the GPU handles each grass billboard (see Section 5.2 and Section 6.3).

Chapter 8

Conclusion

This chapter is a conclusion of the provided diploma thesis concerning GPU based responsive grass containing a short summary and further considerations. Some limitations and future enhancements are finally presented.

8.1 Summary

Real-time applications such as computer games do more and more simulate large natural scenes. Aside from the need of real-time rendered trees, bushes and water there always is a demand for responding grass. To be able to meet the demands thousands of billboards are used to create an illusion of dense grass vegetation. In combination with wind animation nice visual results are achieved. But the visual perception is compromised by lack of interactivity: Objects are moving through the grass without leaving a trace. Due to prior hardware constraints a visually pleasing collision reaction for a large area of grass was unachievable. However, exploiting the potentials of todays GPUs, real-time collision reaction can be achieved as presented in this diploma thesis.

The results of the research in grass simulation are used in order to compose a grass layer on the base of simple quadrilaterals. During the procedural generation process of the grass billboards they are tiled to allow a better handling during runtime. Those billboards are made available to the GPU as point lists. Furthermore, an animation technique for vegetation is employed to the grass layer in order to simulate wind movement. The resulting animated grass layer is presented in chapter four of this diploma thesis.

Since prior research based on the simulation of grass is limited to rendering and animation aspects implementation strategies from other fields of research are consulted. Approaches concerning particle based collision handling as well as real-time cloth simulations are employed in order to implement a collision system as introduced in chapter five. The collision pipeline is split into two subsequent passes: A CPU-based process excludes grass tiles that are obviously not affected by any object collision. Within the geometry shader, implicit object representations are employed to detect penetrations and compute a collision response on the base of each vertex. As the separate processing of individual

vertices may lead to visually unpleasant distortions, a cloth model is evaluated to preserve the overall shape of the grass. After a collision has taken place, the GPU-based collision handling recovers the simple quad shape of the deformed billboards. The restrictions of the stream output stage make it necessary to spread out the billboards data over two point lists.

Illumination as a major element of every realistic landscape-imitation was presented in chapter 6. To integrate the grass layer into a dynamic global lighting environment, precomputed irradiance volumes are employed. Tri-linear interpolation inside the volume allows a computation of incident light for each vertex of the grass billboards approximating global illumination for each vertex. The Gouraud shading is applied during the assembly of the quad primitives. Semi-transparent shaded pixels are accomplished by multiplying the decal color with the reflected incident light. Alpha-to-Coverage blending concludes the process.

The results presented in chapter seven proof that the generation of GPU-based responsive grass (in real-time) is no longer an insolvable challenge. The visual quality in case of dense vegetation and the good performance achieved give a proof of the great suitability of the implementation strategies for achieving large responsive grass layers in todays real-time applications.

8.2 Further Considerations

This diploma thesis presents a collision pipeline which swaps data from one buffer to another while responding to collisions. The collision test as well as the response is based upon each vertex. A subsequent step restores the shape of the mesh. This process can easily be extended as described in the following section. Additionally, the used techniques might be applicable for other plants such as bushes, herbs and crop.

The clearly separated and modular design allows the techniques to be integrated into a large range of real-time applications. Furthermore, as the billboard-based approach is a common way to model grass layers in todays computer games, these implementations can easily be adjusted. However, applications that are already GPU bound will not profit from this approach.

8.3 Limitations and Future Work

The approach presented throughout this diploma thesis grants the availability of responsive grass for dense covered landscapes. The results are demonstrating that collision response works fine for regions where the flat structure of the grass billboards is hardly made out (see Section 7.1.1). However, in areas where grass is planted sparsely, for example at the borders of the grass layer, due to the coarse tessellated mesh of the billboard the visual impression of the deformed billboards is insufficient. Two different approaches, which might also be combinable, might be promising when trying to solve this problem: On the one hand the collision handling could be made independent of the data available for a single primitive. In that case the collision handling for each billboard would be distributed over

several streaming passes which allow for more refined meshes. On the other hand the quad mesh of the deformed billboard can be improved by using an interpolation of a higher degree.

8.3.1 Distributed Spring Relaxation

The billboard's collision response essentially works on each vertex. If a collision occurs, the topology information is absolutely necessary in order to preserve the shape of the grass (see Section 5.2.3.6). However, as a spring affects only neighboring vertices, the collision response can be distributed to several passes:

Initially, the collisions are resolved during a vertex shader pass.[1] As a result of the per vertex based collision response unpleasant distortions of the shape may occur. Subsequently, the cloth simulation model is applied similar to [Zel07]:

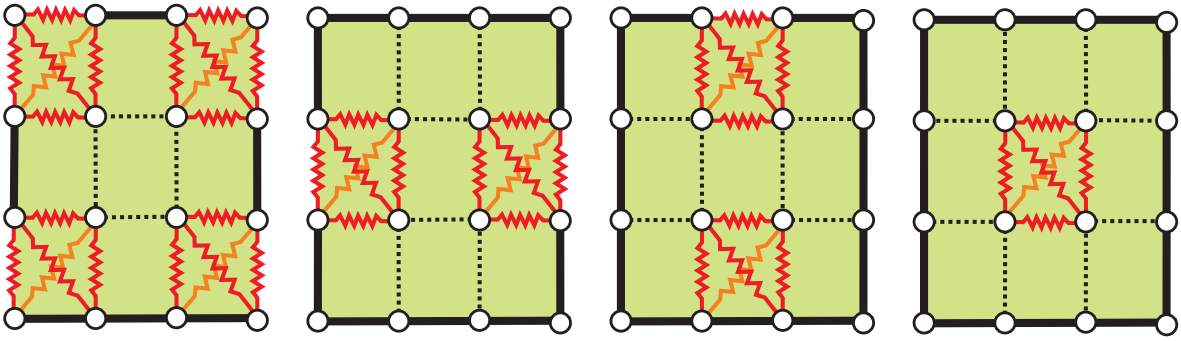


Figure 8.1: The primitive independent spring relaxation. The figure shows the relaxation steps for the quads of a billboard. During a shader pass no relaxed quad may share a vertex with it's adjacent quads. As a consequence four relaxation steps for a billboard are needed.

Four index lists are covering all spring constraints of a billboard as shown in Figure 8.1. The spring mesh of a billboard is relaxed in four streaming passes based on quads. During a geometry shader pass a vertex is only subject to a single quad (one line with adjacency). In other words, those quads are relaxed in a render call that does not share a vertex. Consequently, the springs are relaxed in four steps covering disjunctive quads. After the application of each relaxation pass the unaffected vertices have to be streamed in order to avoid a distribution of the current billboard vertices over both vertex buffers. Finally, the collisions are resolved and the shape of the grass billboard is preserved.

It needs to be pointed out that this process causes some additional work on the CPU. The index lists have to cover all distorted grass billboards in order to avoid a separate relaxation for each billboard. Thus, the lists have to be created dynamically as the amount of collided billboards varies. Moreover, the number of system calls is enlarged.

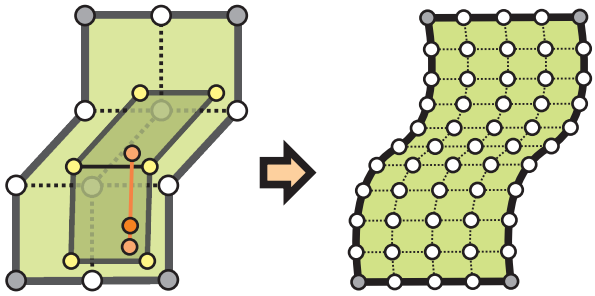


Figure 8.2: The curve based assembly. The figure shows the interpolation scheme for an Bézier surface adopted for the grass billboard. The grass billboard requires different degrees of interpolation along different directions: First each of the six quads is interpolated bi-linearily. The two resulting quads are interpolated bi-linearly as well. A linear interpolation between the resulting points yields the final interpolated vertex.

8.3.2 Curve based Primitive Interpolation

The billboard's deformed mesh consists of six quads (see Section 6.3.2.3). If the assembled primitives are rendered directly, the billboard's shape looks unnaturally angular. A Bézier interpolation between the mesh's vertices might obtain smoother results, as is shown in Figure 8.2: The 4×3 control points require different degrees of interpolation among each of the billboard's dimensions [MH99]. The parameter range is partitioned into numbers of quads which are necessary for the resolution desired. As a result the mesh regarding to the number of quads is much smoother. Texture coordinates and irradiance information are interpolated as well.

As the angular look of the deformed billboards is only visible from a very close position, an additional distance based level of detail technique should be applied in oder to avoid interpolation of the billboards further away from the viewer.

[1]Note that some information has to be stored per vertex. E.g. the index of the vertex is needed in order to compute the wind animation with respect of the vertex location in the mesh.

Appendix A

GPU-Based Distance Maps

As GPU-Based distance maps, sometimes called depth maps, are used for both the collision test as well as for the occlusion term during the generation of the occlusion volume, in this section a short overview of them is presented. Further information can be found in [KLRS04] or [VSC01]. Implicit representations like distance maps provide distances to any point within the distance maps unit view volume. Hence, the distance information can directly be used for penetration tests respectively occlusion tests among objects. Moreover distance maps are especially of importance for GPU based computations. On the one hand the GPU offers the advantage to easily generate them during one single render call; On the other hand the GPU is optimized for texture lookups.

As a first step the projection space is defined as described in section A.1. The distance values are obtained by transforming objects to the distance map's projection space (see section A.2). The distance map then allows for distance comparisons as described in the closing section A.3.

A.1 The Projection Transformation

For the generation of the distance map a projection matrix $\mathbf{T}_{WC \rightarrow DM} \in \mathbb{R}^{4\times4}$ is required which transforms world space positions into the distance map's projection space. Therefore, the near and far clipping plane, the width and height of the projection as well as the origin of the NDC space are computed in world coordinates in order to build the transformation matrix. Initially, the projection basis is specified in world space coordinates $\mathbf{x} = (x_0, x_1, x_2)$ and $\mathbf{y} = (y_0, y_1, y_2)$ and $\mathbf{z} = (z_0, z_1, z_2)$ with the restriction that all vectors have unit length and yield an orthonormal basis. Often almost one single projection direction $\mathbf{z}$ is specified. The setup is shown for a random basis in figure A.1. Afterwards, the center of the projection plane $\mathbf{c}$ has to be defined in world space coordinates. If the whole object or scene is covered, the center then is the closest point to them with respect to the projection direction $\mathbf{z}$ and can be determined using bounding volumes. As no point is closer to the projection plane, in that case the near plane value is set to zero [1]. Accordingly the far clipping plane f then is equal to the distance of the projection center to the farthest point of the object respectively scene. The far plane is

[1] Note that orthographic projection is necessary to do so. Perspective projection expects a value greater zero

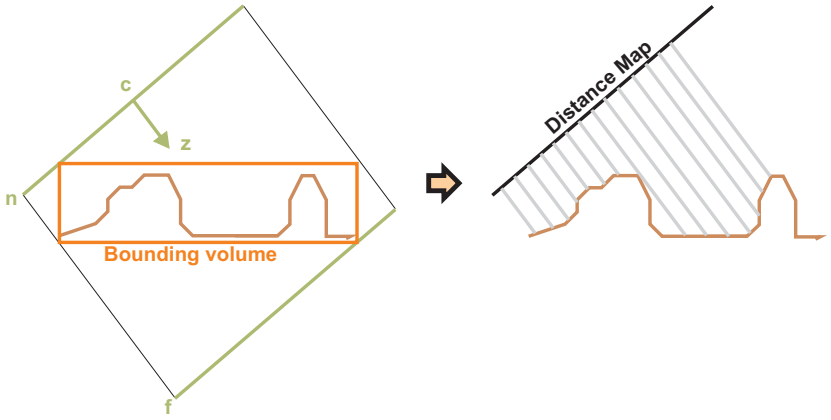


Figure A.1: The distance map setup. The transformation for the orthographic projection is setup by the bounding volume and the projection direction **z**. The generated map then stores relative distances to the scene geometry with regard to the projection space.

set in order to account for the accuracy of the distance values [MH99]. Furthermore the width w and height h of the projection plane is computed. This can be done on the basis of a bounding volume as well, e.g. a bounding box, and is necessary when the whole mesh or scene has to be covered as it is done normally. By using all parameters, the projection matrix is set up (as it is for DirectX):

$$\mathbf{T}_{WC\rightarrow DM} = \begin{pmatrix} \frac{2}{w}x_0 & y_0 & z_0 & 0 \\ x_1 & \frac{2}{h}y_1 & z_1 & 0 \\ x_2 & y_2 & \frac{1}{f}z_2 & 0 \\ -\mathbf{x}\bullet\mathbf{c} & -\mathbf{y}\bullet\mathbf{c} & -\mathbf{z}\bullet\mathbf{c} & 1 \end{pmatrix}, \tag{A.1}$$

where $\mathbf{T}_{WC\rightarrow DM}$ can be thought of as a concatenation of two matrices, a viewing matrix and a orthographic projection matrix.[2] Both then define a view volume in the world space which encloses the object respectively scene.

A.2 The Projection

After the projection transformation $\mathbf{T}_{WC\rightarrow DM}$ is computed, as is described throughout the prior section, the vertices $\mathbf{v} = (x, y, z, 1)$ of the objects then are projected by employing the graphics hardware. Each vertex of those meshes that should be contained in the distance map therefore is transformed using the following transformations:

$$\mathbf{v}' = \mathbf{v}\ \mathbf{T}_{OC\rightarrow WC}\ \mathbf{T}_{WC\rightarrow DM}\ , \tag{A.2}$$

[2] refer to [MH99] for a overview of viewing matrices and projections

where $\mathbf{v}' = (v_x', v_y', v_z', 1)$ is the vertex which is transformed from the mesh's object space to the world space by using $\mathbf{T}_{OC \to WC}$ and then transformed to the depth map's projection space using the transformation $\mathbf{T}_{WC \to DM}$.

In the depth map's NDC space the v_z' coordinate of the transformed vertex then corresponds to the relative distance of the vertex to the projection plane. Thus, the smaller the value of z' the smaller the distance of the vertex is to the projection plane. Next, the vertices $\mathbf{v}'$ transformed to the unit cube are rasterized. The resulting fragments then are used to assign relative distances to each pixel of the 2D distance map at the end of the rendering process. Thereafter, the distance map can be used to read relative distance values $dist(x, y) \in \mathbb{R}$ at the pixel coordinates (x, y) as used during the distance tests as described in the following section.

A.3 The Distance Comparison

After the distance map is filled, the map in combination with the orthographic projection transformation can be used to test for occlusion respectively penetration. The distance information is looked up using the $(x, y) \in [0, 1]^2$ coordinates in the distance map's projection space. Therefore, the point which should be tested either for collision or occlusion is transformed using the same projection transformation applied during the creation:

$$\mathbf{p}' = \mathbf{p}\ \mathbf{T}_{WC \to DM}\ , \tag{A.3}$$

where $\mathbf{p}' = (p_x', p_y', p_z', 1)$ is the projected point and $\mathbf{T}_{WC \to DM}$ is a transformation from the world coordinate space to the projection space with respect of the projection direction. Please note that the point already has to be defined in world space coordinates. Now each point can be tested for penetration: The point penetrates or is occluded whenever the following condition turns out to be true:

$$p_z' > dist(p_x', p_y')\ . \tag{A.4}$$

Before the lookup, the coordinates of the transformed point $(p_x', p_y') \in [-1, 1]^2$ has to be mapped to the distance map's coordinate space $[0, 1]^2$.

Besides, each point that is inside the view volume of the distance maps projection space (see section A.1) after transformation is located in the unit cube $[-1, 1]^3$ of the distance map's projection space.[3] Points that are outside of the view volume before projection mapped to the distance map's border. Sampling the border returns an initial background value. This background value is an extreme large value which does not satisfy equation A.4, thus, the point is not occluded.

[3]Note that DirectX maps the coordinate p_z' to $[0, 1]$ instead of $[-1, 1]$

Bibliography

[BCF+05] S. Behrendt, C. Colditz, O. Franzke, J. Kopf, and O. Deussen. Realistic real-time rendering of landscapes using billboard clouds. *Comput. Graph. Forum*, 24(3):507–516, 2005.

[Ber97] G.vd. Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools: JGT*, 2(4):1–14, 1997.

[BFGS03] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 917–924, New York, NY, USA, 2003. ACM.

[BH02] B. Bakay and W. Heidrich, editors. *Real-Time Animated Grass*, 2002.

[BL06] P. Brown and B. Lichtenbelt. Gl_ext_geometry_shader4, 2006.

[Bly06] D. Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[Bot06] A. Botorabi. *Shader X5*, chapter Animating Vegetation Using GPU Programs, pages 141 – 175. Charles River Media, 2006.

[BPB06] K. Boulanger, S. Pattanaik, and K. Bouatouch. Rendering grass in real-time with dynamic light sources and shadows, 2006.

[Bro08] P. Brown. Gl_ext_texture_array, 2008.

[Bun05] M. Bunnel. *GPU Gems 2*, chapter Dynamic Ambient Occlusion and Indirect Lighting, pages 223–233. Addison-Wesley, 2005.

[BW98] D. Baraff and A. P. Witkin. Large steps in cloth simulation. In *SIGGRAPH*, pages 43–54, 1998.

[CG03] M. Craighead and D. Ginsburg. Gl_arb_occlusion_query, 2003.

[CL07] G. Cadet and B. Lécussan. Fast approximate ambient occlusion. In *SIGGRAPH '07: ACM SIGGRAPH 2007 posters*, page 191. ACM, 2007.

[Dog07] M. Doggett. Radeon hd 2900. In *Graphics Hardware 2007*, 2007.

[Eri04] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, December 2004.

[FGL03] A. Fuhrmann, C. Groß, and V. Luckas. Interactive animation of cloth including self collision detection. In *WSCG*, 2003.

[FS04] D. Fellner and S. Spencer, editors. *Rendering Forest Scenes in Real-Time*, 2004.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[GLM05] N. K. Govindaraju, M. C. Lin, and D. Manocha. Quick-cullide: Fast inter- and intra-object collision culling using graphics hardware. In *VR '05: Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality*, pages 59–66, 319, Washington, DC, USA, 2005. IEEE Computer Society.

[GPR+03] S. Guerraz, F. Perbet, D. Raulo, F. Faure, and M.-P. Cani, editors. *A Procedural Approach to Animate Interactive Natural Sceneries*. IEEE Computer Society, 2003.

[Gra03] Kris Gray. *Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press, Redmond, WA, USA, 2003.

[Gre04] S. Green. *GPU Gems*, chapter Real-Time Approximations to Subsurface Scattering, pages 263 – 278. Addison-Wesley, 2004.

[GRLM03] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In W. Mark and A. Schilling, editors, *Proceedings of the 2003 Annual ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (EGGH-03)*, pages 25–32, Aire-la-ville, Switzerland, July 26–27 2003. Eurographics Association.

[GTGB84] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, 1984.

[HEE+02] M. Hauth, O. Etzmuss, B. Eberhardt, R. Klein, R. Sarlette, M. Sattler, K. Daubert, and J. Kautz, editors. *Cloth Animation and Rendering – Eurographics 2002 Tutorial Notes*, volume T3, Saarbrücken, Germany, September 2002. Eurographics.

[HTG03] B. Heidelberger, M. Teschner, and M. H. Gross. Real-time volumetric intersections of deforming objects. In Thomas Ertl, editor, *VMV*, pages 461–468. Aka GmbH, 2003.

[HTG04] B. Heidelberger, M. Teschner, and M. H. Gross. Detection of collisions and self-collisions using image-space techniques. In *WSCG*, pages 145–152, 2004.

[Hub96] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph*, 15(3):179–210, 1996.

[IC02] J. R. Isidoro and D. Card. Animated grass with pixel and vertex shaders. In Wolfgang Engel, editor, *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware, Plano, Texas, 2002.

[Kaj86] J. T. Kajiya. The rendering equation. In *Proceedings of Siggraph '86*, pages 143–150, 1986.

[KCS07] A. Kharlamov, I. Cantlay, and Y. Stepanenko. *GPU Gems 3*, chapter Next-Generation SpeedTree Rendering, pages 69–92. Addison-Wesley, 2007.

[KHM+98] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[KLRS04] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131, New York, NY, USA, 2004. ACM.

[KP03] D. Knott and D. K. Pai. CIndeR: Collision and interference detection in real-time using graphics hardware. In *Graphics Interface*, pages 73–80, 2003.

[LAM01] T. Larsson and T. Akenine-Moller. Collision detection for continuously deforming bodies, 2001.

[Lan02] H. Landis. Production-ready global illumination. In *Siggraph Course Notes*, 2002.

[LG98] M. C. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In Robert Cripps, editor, *Proceedings of the 8th IMA Conference on the Mathematics of Surfaces (IMA-98)*, volume VIII of *Mathematics of Surfaces*, pages 37–56, Winchester, UK, September 1998. Information Geometers.

[McM03] A. McMahan. *Immersion, Engagement, and Presence - A Method for Analysing 3-D Video Games*, chapter 3, pages 67–85. Routledge Taylor & Francis Group, 2003.

[MH99] T. Moller and E. Haines. *Real-Time Rendering*. A. K. Peters Limited, 1999. In press.

[Mye06] K. Myers. *Shader X5*, chapter Alpha-to-Coverage in Depth, pages 69 – 74. Charles River Media, 2006.

[NV006] Nvidia geforce 8800 gpu architecture overview. Technical report, Nvidia Corporation, 2006.

[NX006] Microsoft directx10: The next-generation graphics api. Technical report, NVidia Corporation, 2006.

[Oat06] C. Oat. *Shader X5*, chapter Irradiance Volumes for Real-time Rendering, pages 333 – 357. Charles River Media, 2006.

[PC01] F. Perbet and M.-P. Cani, editors. *Animating prairies in real-time*, 2001.

[Pel04] K. Pelzer. *GPU Gems*, chapter Rendering Countless Blades of Waving Grass, pages 107 – 121. Addison-Wesley, 2004.

[PG04] M. Pharr and S. Green. *GPU Gems*, chapter Ambient Occlusion, pages 279 – 292. Addison-Wesley, 2004.

[Pro95] X. Provot. Deformation constraints in a mass–spring model to describe rigid cloth behavior. In *Graphics Interface '95*, pages 147–154, May 1995.

[Ram05] R. Ramraj. *Game Programming Gems 5*, chapter Dynamic Grass Simulation and Other Natural Effects, pages 411 – 419. Charles River Media, 2005.

[Sat06] R. Sathe. *Shader X5*, chapter Collision Detection Shader Using Cube-Maps, pages 533 – 542. Charles River Media, 2006.

[Sch06] T. Scheuermann. Render to vertex buffer with d3d9. In *SIGGRAPH 2006 Course 3: GPU Shading and Rendering*, August 2006.

[Sek04] D. Sekulic. *GPU Gems*, chapter Efficient Occlusion Culling, pages 487 – 503. Addison-Wesley, 2004.

[SKS02] P.-P. J. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph*, 21(3):527–536, 2002.

[Sou07] T. Sousa. *GPU Gems 3*, chapter Vegetation Procedural Animation and Shading in Crysis, pages 373 – 407. Addison-Wesley, 2007.

[TKZ+04] M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, and W. Strasser. Collision detection for deformable objects. In *Proc. Eurographics, State-of-the-Art Report*, pages 119–135, Grenoble, France, 2004. Eurographics Association.

[VSC01] T. I. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. *Comput. Graph. Forum*, 20(3), 2001.

[WC08] C. Woolley and N. Carter. Gl_nv_transform_feedback, 2008.

[Wha05] D. Whatley. *GPU Gems 2*, chapter Toward Photorealism in Virtual Botany, pages 7–25. Addison-Wesley, 2005.

[Wil78] L. Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, 1978.

[WNDS99] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Zac98] G. Zachmann. Rapid collision detection by dynamically aligned dop-trees, 1998.

[Zel07] C. Zeller. Cloth simulation. White paper, NVidia, 2007.