FAKULTÄT FÜR !NFORMATIK

Real-Time Rendering of Dynamic Vegetation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Alexander Kusternig
Matrikelnummer 0026571

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Phys. Dr.techn. Ralf Habel

Wien, 08.05.2009

_____ (Unterschrift Verfasser)

_____ (Unterschrift Betreuer)

Alexander Kusternig

Haupstraße 76/18
2372 Gießhübl

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

Alexander Kusternig, Wien, 08.05.2009

Abstract

Plants are present in almost any type of interactive virtual environment like video games, movie pre-visualization or architectural or urban walkthroughs. The simulation complexity of plants increases with the evolution of graphics hardware, but rendering of plants still poses a lot of challenges. This is due to both the inherent geometric complexity of an individual tree having thousands of branches and tens of thousands of leaves, and the complex light interactions between the plant and sunlight. A portion of incoming light is transmitted through leaves, resulting in the bright translucency effect observed when looking at a leaf against the sun. Animating plants is another challenge, as thousands of interconnected branches and individual leaves have to react to turbulent wind moving through the treetop. All this should be performed at more than 60 frames per second for real-time interactive applications.

This thesis presents novel algorithms to render leaves at very high detail with a physically based translucency model and to animate branches and leaves using a stochastic approach based on their physical properties. Both algorithms are executed entirely on the GPU in vertex and pixel shaders, so they can be easily integrated into any modern rendering pipeline. The efficiency of the algorithms allows rendering and animating highly detailed plants with thousands of branches and tens of thousands of leaves at a frame rate of at 60 frames per second.

Kurzfassung

Pflanzen finden sich in fast allen interaktiven virtuellen Umgebungen wie Videospielen, Szenen-Vorberechnungen in der Preproduction von Filmen oder Architektur-Walkthroughs. Mit jeder neuen Generation von Grafikhardware steigt auch die Qualität der Simulationen, aber die inhärente geometrische Komplexität von Pflanzen, die aus tausenden von Ästen und zehntausenden von Blättern bestehen, und die komplizierte Interaktion mit Sonnenlicht stellen immer noch große Herausforderungen dar. Blätter lassen einen Teil des ankommenden Lichts durch, was einen hellen Durchleucht-Effekt erzeugt, wenn sie gegen die Sonne gehalten werden. Die Animation von Pflanzen ist eine weitere Herausforderung, da tausende von miteinander verbundenen Ästen und dazugehörigen Blättern auf Wind reagieren müssen, der durch die Baumkrone streicht. Für Echtzeitanwendungen müssen diese Effekte mit mindestens 60 Bildern pro Sekunde dargestellt werden.

Diese Diplomarbeit beschreibt zwei Algorithmen um Blätter mit hohem Detailgrad und einer physikalisch basierten Lichtdurchlässigkeit darzustellen und um Äste und Blätter entsprechend ihrer physikalischen Eigenschaften zu animieren. Beide Algorithmen werden direkt auf der GPU im Vertex und Pixel Shader ausgeführt und können daher einfach in jede moderne Echtzeit-Renderingapplikation integriert werden. Die Algorithmen sind effizient genug um Pflanzen mit tausenden von Ästen und zehntausenden von Blättern flüssig mit mehr als 60 Bildern pro Sekunde darzustellen und zu animieren.

Contents

Chapter 1

Introduction

Computer graphics applications are ubiquitous in modern technologies, including navigation tools in mobile devices, entertainment in computer and video games or movies, scientific visualizations for medical purposes and simulations or virtual reality walkthroughs. Vegetation is a part of most of these applications. Besides outdoor natural scenes, plants are also important in urban scenarios or indoor scenes where alley trees, potted plants or patches of grass can be found. As graphics hardware evolves, so does the complexity of the simulation of vegetation. The main driving force behind graphics hardware evolution today is the entertainment sector, so real-time constraints as well as convincing representations become increasingly important.

1.1 Aim of the Thesis

This thesis proposes algorithms to render plants useable for real-time virtual environments like games. The results should not only look convincing but also have to be rendered at more than 60 frames per second to allow real-time user interaction. Besides rendering each frame at a high level of quality, the animation of plants caused by wind has to be considered for interactive applications. Figure 1.1 shows a tree rendered with the techniques presented in this thesis.

The algorithms introduced in this thesis are designed for easy integration into any modern rendering pipeline because they run only in the vertex and pixel shaders of an application. No additional computations on the CPU are necessary at runtime. The techniques are resource-efficient and fine-tuning of the appearance of the resulting image can be controlled by a small set of intuitive parameters.

Fig. 1.1: A tree rendered with the algorithms presented in this thesis.

1.2 Challenges

There is a number of challenges related to the convincing representation of plants in real-time computer graphics:

Geometric complexity

Plants have a high geometric complexity. An individual tree can have thousands of branches and tens of thousands of leaves. Each of these have to be rendered at high detail for viewpoints close to or inside the plant. There is a vast body of research on the generation of geometric data to represent plants. While plant growth in nature is governed by basically the same rules for all types of plants, very different forms result from it. This also leads to various methods to simulate plant growth mostly using L-systems [57] or procedural techniques.

Besides the generation of hierarchical branch data, the geometric *simplification* is a main topic of many publications. Simplification is needed for LOD (Levels of Detail) techniques where a large number of plants have to be rendered at the same time, for example in a forest scene. Most of these

plants only cover a small area of pixels on the screen, so rendering them at their full geometric complexity is neither feasible for real-time applications nor for offline rendering.

Light Interaction

The interaction between light coming from the sun or light reflected by the environment and the plant is of a very complex nature: Leaves have a strongly structured surface with risps, bulges and veins. Also, they consist of multiple layers of tissue which react differently to incoming light. Subsurface scattering effects occur as rays of incoming light are deflected between these layers until they finally exit the leaf on either side. Light leaving on the opposite side leads to a noticeable *translucency* effect. Leaves transmit up to 20% of the incoming light. Light gradually diminishes with each additional layer of leaves it passes, so *indirect* and *scattered* illumination is the dominant illumination source inside a treetop.

Convincing Animation

Finally, plants should be rendered not only for still images but for real-time interactive applications. This introduces the influence of *wind* onto a plant, resulting in turbulent branch and leaf motion. Even slight wind causes small twigs and leaves to react to it, resulting in continuous swaying motion. A treetop is almost never completely motionless. Stronger wind on the other hand noticeably bends a whole plant into the wind direction. Wind reaction depends on the physical properties of each branch, including its thickness, length, elasticity, and where other branches and leaves are attached to it. This makes wind interaction a highly complex process.

Existing work on animation is usually based on a hierarchical representation of the plant. Either the hierarchical elements are influenced by a full simulation or stochastic processes are used to recreate the motion of branches and leaves seen in nature.

Rendering at 60 frames per second

Rendering these plants at high detail in *real time* for interactive applications poses the additonal challenge of each frame having to be rendered within 16.7 milliseconds if a stable frame rate of at least 60 images per second is to be maintained. This is in strong contrast to offline rendering for movies, which allow each frame taking hours to render. Optimization and precomputation becomes a necessity when dealing with real-time constraints.

1.3 Scope of this Thesis

This section highlights which of the challenges described in the previous section are dealt with in this thesis:

Leaf rendering

A technique is presented to render the complex appearance of leaves including detailed microstructures and translucency. The physical properties of leaf surfaces and bodies are taken into account to simulate translucency as a subsurface scattering process of sunlight transmitted through the leaf and exiting on the opposite side. It is possible to move the camera at the distance of only a few centimeters to the surface of a leaf and still gain a highly detailed image. The algorithm is executed completely in the pixel shader, no computations on the CPU are necessary at runtime. Figure 1.2 illustrates the translucency effect of light transmitted through a leaf. Also the data acquisition process is described, which uses off-the-shelf hardware to create the textures needed as a basis for this technique.

Fig. 1.2: Leaf translucency can be observed when looking towards the sun.

Animation

An animation algorithm is introduced which allows animation of complex plant models. Each branch and each leaf is animated separately, based on a stochastic process which relies on the physical properties of branch motion interacting with turbulent wind moving through the treetop. The algorithm is executed only in the vertex shader, so a non-linear deformation of branches can be achieved as they get thinner towards their free end, and the complexity of the algorithm is not limited by the number of branches or leaves in the model, which is a great improvement over exiting methods.

Geometric complexity

However, this thesis does *not* deal with the geometric complexity posed to render plants at various levels of detail. Both the leaf rendering algorithm as well as animation are executed solely on the GPU in the pixel shader and vertex shader respectively so it can be assumed that these techniques scale linearly with the number of pixels which a plant covers on the screen on the one hand and with the number of triangles used to represent it geometrically on the other. Reduction of geometric detail is possible while still retaining full rendering and animation quality, so the presented techniques comply perfectly with LOD methods.

1.4 Contribution

The ideas for a physically based leaf translucency algorithm and a physically guided tree animation model were developed by Ralf Habel. These ideas were extended to algorithms composed of offline pre-processing and a runtime component in GPU vertex and pixel shaders in cooperation between the author and Ralf Habel.

The author implemented the algorithms and embedded them in a real-world state-of-the-art rendering system, which does not only consist of an implementation of the two proposed methods but also includes high dynamic range rendering, shadow mapping, and advanced per pixel lighting techniques. The algorithms have to perform at least at 60 frames per second while still leaving enough rendering resources for other parts of the pipeline. The application also allows testing performance characteristics with different geometric or algorithm complexities and rendering pipeline configurations.

The offline tools to generate necessary data for rendering and animation were created in cooperation between Ralf Habel and the author. Taking photographs and 3D scans of leaves as well as generating tree geometry in *XFrog*

was done by Ralf Habel. The algorithms were published in *"Physically Based Real-Time Translucency for Leaves"* and *"Physically Guided Animation of Trees"*, respectively [29, 30].

1.5 Thesis Structure

The thesis is structured as follows:

- Chapter 2 presents state-of-the-art techniques for leaf rendering.
- Chapter 3 introduces a novel approach to render leaves at high detail, including a physically-based translucency model.
- Chapter 4 lists state-of-the-art techniques to handle animation of trees.
- Chapter 5 shows a new animation algorithm to animate trees of very high geometric complexity based on a stochastic process.
- Chapter 6 describes the demo application created to test the algorithms proposed in this thesis and presents results.
- Chapter 7 sums up the contents of the thesis and provides a glance at further extensions to the algorithms.

Chapter 2

State of the Art on Leaf Rendering

Both the amount of light which is reflected off the leaf surface and the light transmitted through a leaf have to be taken into account to generate convincing images. A sophisticated light transport solution has to be found, as leaves are constructed from multiple layers of tissue with different optical properties. Subsurface scattering is a process which occurs if the light is deflected between these layers until ultimately exiting the leaf on either the top or the bottom side. Section 2.1 discusses subsurface scattering in general.

All publications which deal with leaf rendering have to take subsurface scattering into account, although very different approaches are taken. There are models which use ray tracing to trace through a geometric representation of these layers of tissue as well as methods based on evaluating a Bi-directional Scattering Surface Reflectance Distribution Function (BSSRDF) [49] constructed either analytically or from measured data. Section 2.2 describes subsurface scattering in respect to leaf rendering.

Section 2.3 presents a series of publications using radiative transfer methods to simulate the light distribution in plants. Ray tracing methods are used as well as spherical harmonics to evaluate the amount of light exiting and leaving each surface point analytically. However, most of these methods are not suitable for real-time applications, either because of their computational complexity or because of their memory consumption.

Section 2.4 introduces diffusion-based methods which simulate subsurface scattering as a diffusion effect. Leaves are treated as thin slabs of homogeneous material which allows simplifications to the BSSRDF and instancing of reflectance and transmittance data for multiple leaves. These methods prove to be suitable for real-time rendering applications.

2.1 Subsurface Scattering

Subsurface scattering (SSS) is an important research field in computer graphics. It deals with the interaction of light with scattering media. Light nor-

mally is not entirely reflected off the surface, but a certain amount of light enters the medium and is scattered in a range of usually a few millimeters beneath the surface until exiting again at another point on the surface. For some materials like wax the effective depth of scattering is even multiple centimeters. The effects of subsurface scattering can be seen as smoother gradients on objects as well as the effect of highlighting of thin parts of the object or "rim lighting" at grazing angles. Skin rendering is one of the most popular applications for subsurface scattering simulations [16]. Subsurface scattering has been expanded taking more general lighting conditions into account [75] or applied onto deformable objects [43].

Usually objects considered for subsurface scattering are placed inside a non-scattering medium like air. In this case, scattering can be fully described using a Bi-directional Scattering Surface Reflectance Distribution Function (BSSRDF) [49]. Additionally to the already 4-dimensional Bidirectional Reflectance Distribution Function (BRDF) the point where light enters the object and where is exits do not have to be the same. This makes the BSSRDF a *non-local* function and increases the dimensionality to 8. A full BSSRDF is very costly to evaluate. It can be done accurately by path tracing, but methods used in research usually simplify the model and reduce the number of dimensions.

2.2 Subsurface Scattering in Leaves

Only a small number of publications model SSS specifically for leaves. Leaves pose additional problems on subsurface scattering evaluation. For example with skin rendering, light is assumed to exit the medium on the same side of the medium on which it enters. However, with a thin translucent medium like a leaf, light could also exit on the *opposite* side. This is what creates the translucency effect of light transmitted through a leaf when looking at it against the sun. A sophisticated subsurface scattering solution is crucial mostly for the translucent sun-averted back side of a leaf to capture the effects seen in nature.

For the sake of simplicity, most publications regarding subsurface scattering assume the scattering parameters to be homogeneous over the whole surface [23, 74]. With leaves the detailed microstructure of the surface formed by risps and bulges and the structure of the interior - including a varying thickness - increase the complexity of the simulation further. The locally variant surface and interior composition of a leaf prohibit the use of a generic solution for the whole medium and methods have to deal with spatially varying attributes.

This also leads to most publications concerned with leaf rendering using *measured* data to deal with varying reflectance, translucency, and thickness properties instead of generating the subsurface scattering solution *synthetically* by fitting it to a formula. The first step in rendering subsurface scattering is to acquire the necessary underlying data, which is described in the next section.

2.2.1 Measurements

Spectro-Photo-Goniometers can be used to capture the bidirectional reflectance and transmittance directly [73]. Due to the nature of these instruments capturing detailed *spectral* information but only low-frequency *spatial* data, the measurements are averaged over a larger area of the leaf surface and do not incorporate the detailed spatial variances in surface structure [10]. Of course, photographed textures can be used to modulate the reflectance and transmittance characteristics, but subsurface scattering is not correctly accounted for.

Wang et al. [74] use a Linear Light Source (LLS) [24] as seen in Figure 2.1 to measure the diffuse color, specular color and specular roughness of a leaf on a per-pixel basis. In contrast to the method mentioned previously, a LLS allows the generation of high-resolution texture maps which capture all the surface detail. These maps capture both the BRDF and Bi-directional Transmittance Distribution Function (BTDF) behavior for both sides of a leaf. The main problem however is that a LLS is not commonly available and has to be custom-built for this purpose.

2.3 Radiative Transfer Models

Most publications concerned with leaf rendering are based on calculating the radiative transfer of light, which is the amount of light hitting and leaving a surface point. Radiative transfer presents the possibility to evaluate direct, indirect and environment lighting in a unified model, but needs extensive precomputation. Additionally, the generated data is unique for each evaluated surface point, resulting in high memory consumption. Usually, radiative transfer models can not be evaluated in real time if done on such a micro-detail scale as is needed to capture subsurface scattering.

2.3.1 Ray Tracing

As one of the first publications dealing with realistic leaf rendering Hanrahan et al. [34] used Monte Carlo ray tracing to evaluate the subsurface scattering

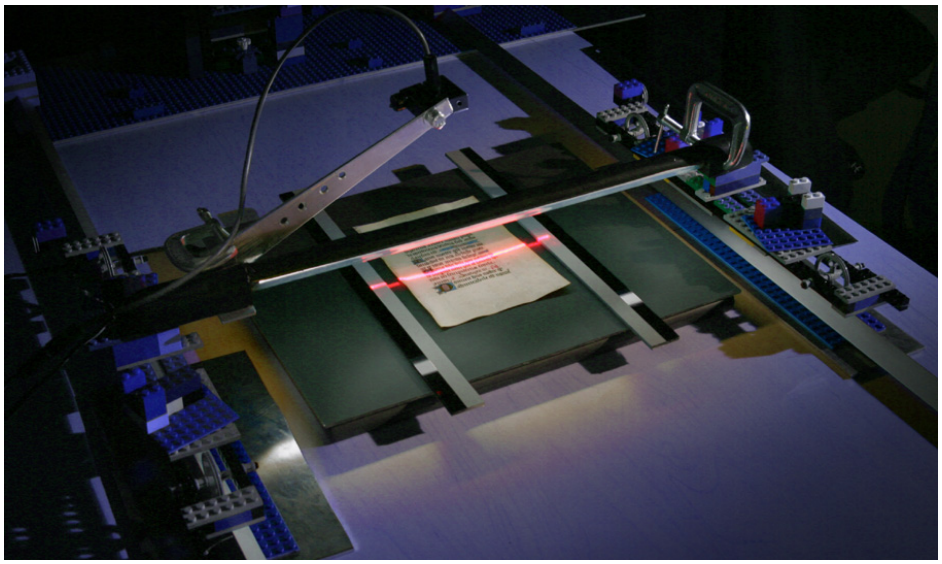

Fig. 2.1: A linear light source used to measure reflectometry.

solution. One-dimensional linear transport theory was employed to derive an explicit formula for reflectance and transmittance. This does not account for multiple scattering, though, and does not include a full BSSRDF approach.

Govaerts et al. [27] model the interior of a leaf as a detailed geometrical representation. Different tissues with varying optical properties are explicitely modeled. Ray tracing is applied to trace through all of these layers. Obviously, this is slow and not suitable for real-time rendering systems. Baranoski et al. [4] later expand this model by using available biological information. A simplified scattering model by Baranoski et al. in 2001 [5] uses this model to precompute reflectance and transmittance values for the leaf surface. The model is controlled by a number of biologically meaningful parameters such as pigment concentractions, refraction index and oblateness of epidermic cells. The model proves useful to predict specular reflectance properties of leaves. Figure 2.2 shows Baranoski's results.

Ganapol et al. [23] propose the *LEAFMOD* model to calculate estimates of leaf reflectance and transmittance. The model solves the one-dimensional radiative transfer in a slab with homogeneous optical properties. Still, the full BSSRDF is not taken into account. The leaf is assumed to be of uniform thickness and of a homogeneous material. Local variations, resulting from the mesostructures and self-shadowing of those structures is not considered.

All methods based on ray tracing are extremely computationally expensive, since a large number of rays have to be shot per pixel to approximate the subsurface scattering solution accurately. While these methods are obvi-

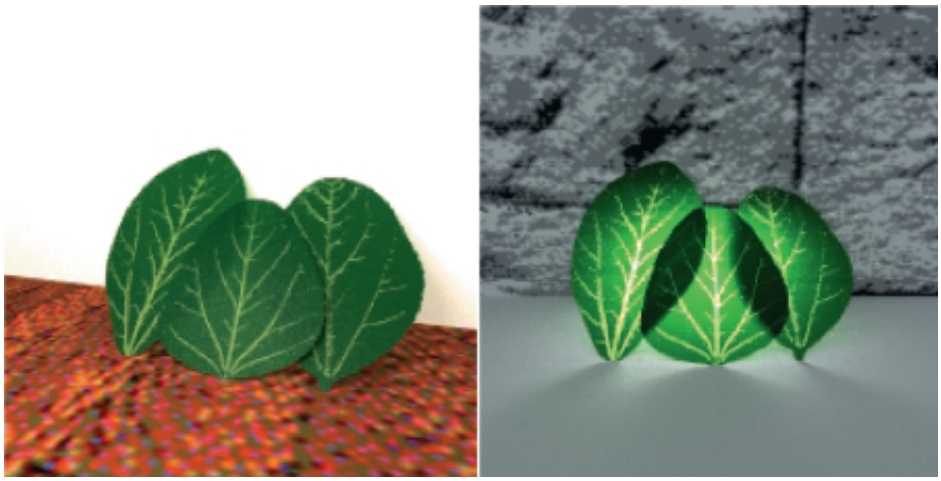

Fig. 2.2: A set of front-lit and back-lit leaves rendered by Baranoski et al.

ously not suited for real-time rendering, they prove even tedious when used only for preprocessing of data.

2.3.2 Radiative Transfer for Real-Time Rendering

Wang et al. [74] make use of the LEAFMOD model. They obtain their input data by a LLS, measuring diffuse and specular reflectance, varying roughness as well as diffuse transmittance for both sides of a leaf at a highly detailed scale of sub-millimeter accuracy. Fitting this data into the LEAFMOD model gives them a thickness variation map, an albedo map and an average absorption and scattering coefficent. Having all this detailed data available in the form of high resolution texture maps allows the evaluation of reflectance and transmittance in a real-time environment. The model proposed by Wang et al. takes direct, indirect and environment lighting into account. Because all the lighting data is precomputed and stored in texture maps, certain simplifications are made to keep the amount of memory low. A unified precomputed radiance transfer solution could be used to evaluate the whole lighting for each surface point, but a simpler model is used. Lighting calculation is split up into two parts:

- *Direct illumination* depends on the amount of sunlight hitting the surface point directly. Possible sunlight visibility is precomputed for each vertex as an environment map. The amount of direct lighting is then evaluated by convolution of this environment map with a sun disk projected to a corresponding environment map. This delivers the hard shadows usually observed in direct sunlight illumination.

- *Indirect and environment lighting*, which are both of relatively low frequency when compared to direct illumination, are approximated using spherical harmonics (SH) [65]. The precision of a SH solution depends on the number of coefficients used. To evaluate the SH solution the incoming light has to be transformed into SH coefficients as well. The big advantage of SH is that not only a simple direct light source may be used, but an arbitrary low-frequency environment lighting can be expressed by SH coefficients.

  Usually, 16 coefficients are used because they fit perfectly into 4 vectors of 4 components each, and can therefore be optimally stored in 4 vertex attributes or textures. Due to the limited number of coefficients, spherical harmonics tend to produce soft lighting, which is well suited to simulate indirect and environment lighting, but has problems with the hard shadows coming from direct sunlight. A larger number of coefficients is needed to approximate direct illumination, which further increases the amount of memory needed. The low-frequency nature of SH combined with the memory consumption is the reason Wang et al. decided to use this simulation for indirect illumination only.

Wang et al. evaluate lighting on a per-vertex basis, and interpolate the results. So, since all of the illumination data is pre-baked and data is unique per vertex, the precision of the solution depends directly on the geometric complexity of the mesh. This method is still very memory-expensive, Wang et al. only model small plants with less than 20 leaves. They report that these plants already consume about 30 MB. While producing impressive results as can be seen in Figure 2.3, this model proves not to be feasible for larger plants due to memory constraints and vertex count limitations. A leaf still needs to be rendered at full geometric detail even if the leaf is far away from the viewer.

Also the surfaces of the leaves appear to be very smooth, which is only valid for some types of leaves. This is due to using the geometric normal on a per-vertex basis for lighting evaluation instead of the high-detail surface normal which is available from the data measured by the LLS. However, evaluating spherical harmonics on a perturbed normal requires additional computation, as Sloan et al. [64] demonstrate.

Also, precomputation of all lighting data prohibits the plant from being animated. Methods exist for animating spherical harmonics but result in an additional workload on memory and computation time [66]. The method described in this thesis aims for both memory efficiency as well as animating plants by wind influence, so relying on precalculated lighting only is no suitable approach.

Fig. 2.3: Three plants modeled with the technique presented by Wang et al. All plants consist of only a small number of leaves to keep memory requirements low.

2.4 Diffusion-Based Methods

Jos Stam [68] treats light scattering in a medium as a diffusion process. This is true for highly scattering media for which full path tracing is not necessary. Stam uses a multi-grid approach to render multiple scattering in clouds. Jensen et al. [35] model an analytic expression based on dipole diffusion approximation of a semi-infinite thin slab. Again, the material is assumed to be homogeneous to avoid evaluating a detailed BSSRDF for every point on the surface. Mertens et al. [43] show that this model can be integrated into a real-time rendering environment.

Donner and Jensen [19] extend the model to efficiently calculate subsurface scattering in multi-layered thin slabs by using multiple diffusion dipoles. This approach adds an additional boundary on the opposite side of the medium to accurately compute the amount of light *transmitted* through it. Obviously, this is needed to calculate reflectance and transmittance for leaves. Due to their thin nature, the influence of subsurface scattering is constrained to a small surface area, and full evaluation over the whole leaf surface is not necessary to calculate the subsurface scattering solution. This significantly reduces precomputation work, and also allows the assumption of the BSSRDF solution to be considered *local*. This is a very important assumption because the space for evaluating the BSSRDF has to be locally flat. This would pose problems with bent leaves if the whole leaf had to be taken into account. Also, because the evaluation stays local the same BSSRDF can be used for all leaves. There is no need to calculate a new BSSRDF for each leaf, which keeps memory requirements low for this method.

Figure 2.4 shows the quite realistic results of the method. Different reflectance properties can be observed for the front and back sides of the leaf. The transmittance on the other hand seems to be nearly identical for both sides. These observations correspond to measured data and prove the accu-

racy of the method.

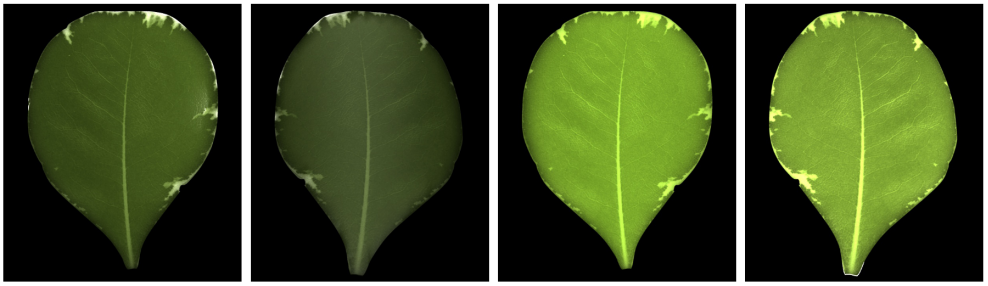

Fig. 2.4: The front and back sides of a leaf. Rendered with light coming from the front (left), and with light coming from the back (right).

Franzke and Deussen [22] use a simplified model of Donner and Jensen's method by reducing the leaf to only one layer and evaluating only single scattering in the light transport. Ignoring multiple scattering effects significantly reduces the rendering cost. The results of Franzke can be seen in Figure 2.5.

Fig. 2.5: A plant rendered by Franzke and Deussen. Only single scattering is taken into account.

The multiple-dipole approach proposed by Donner and Jensen was used by the technique presented in this thesis to precalculate a local illumination environment for a leaf taking a BSSRDF and surface mesostructures into account, as can be seen in Section 3.5. However, the HL2 basis was chosen as a simpler method to evaluate this data in a real-time constraint, and leaves are assumed to consist of only one slab.

Chapter 3

A Real-Time Leaf Rendering Algorithm

This chapter presents an algorithm to render photorealistic leaves with a physically based translucency model in a real-time environment. The runtime part of the algorithm is based on vertex and pixel shaders only, so it can be integrated into any modern rendering engine. Precomputation of necessary information is done offline in separate tools, and the information is passed to the shader either as per-vertex data or as texture content.

The algorithm was designed to allow two important features: First of all, leaves have to retain a believable high-detail look even if viewed at very close range. Even if one single leaf covers the full screen, details should stand out clearly and the mesostructue of risps and bulges on the leaf's surface and veins in the leaf's interior are visible. Second, the method should scale well to a large number of leaves and to multiple plants. To achieve this, data must be stored efficiently and be reuseable as much as possible.

3.1 Overview

To evaluate the full lighting situation for a pixel the illumination of direct sunlight, indirect sunlight, and environment lighting have to be taken into account. Only direct sunlight is evaluated by this method on a per-pixel basis at runtime, indirect illumination is approximated by a precomputated solution. The leaf rendering algorithm described in this thesis consists of two major parts:

- *Direct illumination* is handled on a per-pixel level and captures the high-detail mesostructure of leaf surfaces. Normal mapping is used to express the structures. Shadow mapping is used to handle the hard shadows resulting from direct illumination. All of the data needed for evaluation of the direct illumination is available locally by texture data, and the same data set is instanced for every leaf in a tree. The only exception to this is the shadow map, which is global and generated

at runtime each frame. The components of direct illumination are described in detail in Section 3.3.

Translucency and subsurface scattering effects are handled by a precomputed radiance transfer solution which is based on the "Half Life 2" (HL2) basis, and also stored in a texture. The HL2 basis can be set up to match with local leaf space so the same texture can be instanced for all leaves in a tree. The tangent space – which has to be built anyway for normal mapping – is also used to define the basis vectors of the HL2 space. Details on translucency can be found in Section 3.5.

- *Indirect illumination* handles the transport of environment light and indirect sunlight through different layers of leaves down to the branches or the trunk. This illumination component is precalculated once for the whole tree and illumination information is stored on a per-vertex basis. Details on the calculation of the indirect illumination term is found in Section 3.4.

Figure 3.1 illustrates how direct and indirect illumination together create the final appearance of a leaf.

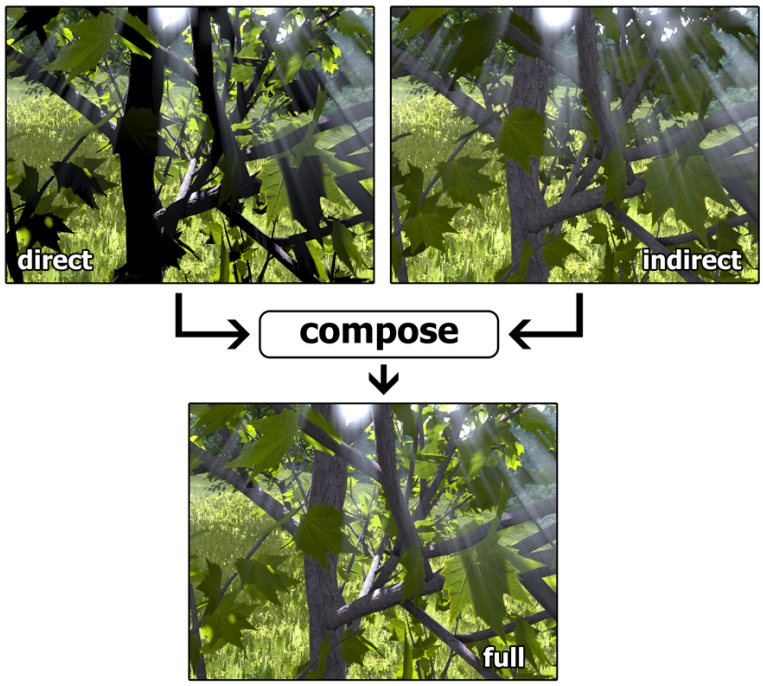

Fig. 3.1: Direct illumination only (top left), indirect illumination only (top right), and the composed final image (bottom).

Section 3.2 of this chapter deals with the acquisition of data needed to simulate all of these effects. Texture sets for leaves were generated by photographing leaves and lighting them from the front to obtain an albedo map and from the back to gain the average translucency color information. Additionally, 3D scanning was used to gain information about the structure of the leaf surface as well as to obtain per-pixel thickness information to calculate translucency and subsurface scattering information.

3.2 Data Acquisition

The rendering application makes use of a leaf acquisition method developed by Ralf Habel which will be described in this section.

The front and back sides of a leaf were photographed two times each. First with light coming from diagonally behind the camera to gain an albedo map to capture the diffuse reflectance properties of the leaf. A second photo was taken with the light source placed behind the leaf, to obtain translucency color information of light transmitted through the leaf.

Additionally, the leaf was placed into a 3D scanner to get a high-detail geometric representation of the leaf. The data sets of the front and back sides were used to calculate the normal maps and a thickness map of the leaf. While the normal map is used directly for normal mapping, the thickness map is needed along with the normal map to generate subsurface scattering and translucency information which is stored in the HL2 map. Details on the acquisition of the HL2 map are found in Section 3.5.4.

3.2.1 Photographing Leaves

A Canon EOS 20D digital camera was used to capture the photographs for a leaf set. The images were taken at a resolution of 3504*2336 pixels in RAW format with fixed exposure time. Two large box diffusors with a 1000 Watt light source each were used to light the leaf from the front, with the diffusor placed next to the camera, and from the back, with the second diffusor placed behind the leaf. The diffusors create almost hemispheric lighting for front and back lighting, which is needed to get the diffuse reflectance information. Strong directional light sources could result in shading of the leaf surface as well as intense specularity. The leaf itself was placed in an easy-to-build wooden fixing frame to ensure that all photographs match pixel-perfect, and to reduce distortions due to the natural bending of the leaf. A photograph of the installation together with a schematic view of the acquisition setup can be seen in Figure 3.2.

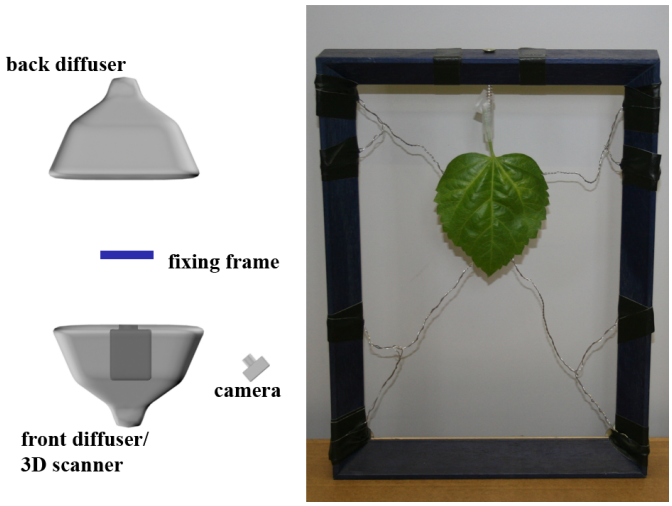

Fig. 3.2: The photography setup used to take photos of the leaves. The leaf is placed inside a fixing frame to ensure pixel-exact matching between photos. Image courtesy of Ralf Habel [29].

After taking photographs of one side of the leaf the fixing frame was carefully rotated by 180 degrees to capture images of the other side. The resulting images were downsampled to 1024*1024, which still retains high-resolution color information in the sub-millimeter range. Specular highlights still present were removed from the photograph with standard image processing tools to obtain only diffuse color information for the albedo map. Also, an alpha channel was added to this texture for alpha-testing when rendering the texture mapped onto a quad. At a total there are four textures obtained from photographs: An albedo and a translucency color map, both for the front and the back side of a leaf.

3.2.2 3D Scanning

A Minolta VI-910 scanner was used to capture the 3D geometry of both the front and the back sides of a leaf. As with the texture maps, the resolution of the scanned geometry is within sub-millimeter range. High resolution normal maps are obtained directly from the geometry. These two scans were combined into one geometric object with the help of Geomagic [25], and scanning errors were smoothed out. Autodesk Maya [3] was used to calculate a normal map for each side and generate different lower-resolution meshes of the original geometry. The photographed textures are then mapped onto them.

The thickness map is calculated from measuring the difference between

the front and the back side of leaf geometry and normalising the differences to a user-defined maximum. Figure 3.3 shows all of the textures for one leaf data set, downsampled to 1024*1024 pixels.

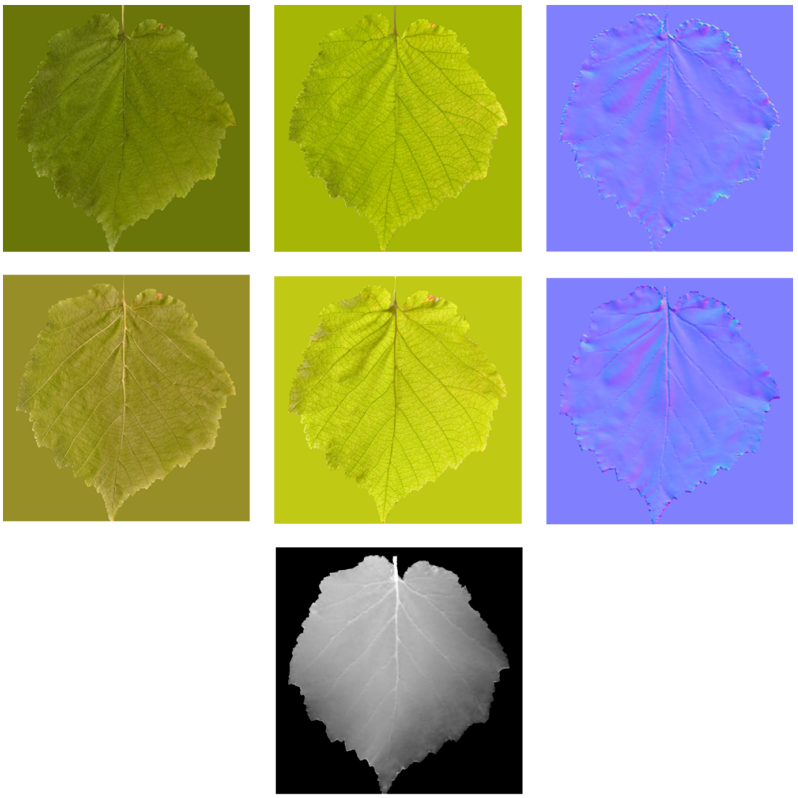

Fig. 3.3: A data set for albedo maps (left), translucency color maps (middle) and normal maps (right) for the front and back sides of a leaf, and the thickness map (bottom). Image courtesy of Ralf Habel [29].

3.3 Direct Illumination

Direct illumination is evaluated in the pixel shader. The final color value consists of multiple parts: An albedo map is used for the basic per-pixel reflectance color, normal mapping is used to deal with the detailed structure of the leaf surface. Shadow mapping is used to determine sunlight visibitily.

The Cook-Torrance specular model [14] is used to calculate the specular reflectance. Typically, only the top side of a leaf is specular, while the bottom side is mostly diffuse [10]. The Cook-Torrance specular model is

controlled by physical parameters like surface roughness and takes a Fresnel term into account, which is an important part of the specular appearance of leaves. The specular highlights of leaves tend to disperse and stand out more clearly if seen at grazing angles. The benefits of this model become obvious when compared to the "standard" specular model for real-time rendering, the Phong-Blinn specularity model [8]. That model usually creates continuous bright spots of specularity, a behavior not found for most types of leaves.

The last part of the direct illumination model is the translucency, which simulates the amount of sunlight transmitted through the leaf and exiting on the opposite side. This effect causes the perceived bright green appearance of leaves if looked at against the sun. This property is also seen from a distance, which is not the case for the high-detail techniques like normal mapping or shadow mapping, which could be simplified at a distance to reduce rendering cost. Another physical property which comes with translucency is subsurface scattering. Rays of light which are not reflected by the topmost layer of a leaf surface are bounced off or scattered between multiple interior layers until they exit the medium again. In the case of leaves this may happen on either side. Light emitted on the incoming side adds to the reflectance while light emitted on the opposite side adds to the translucency. Subsurface scattering is handled by a precomputed radiance transfer approach which is calculated once for a leaf texture set. The simulation is local to the leaf only, not taking into account global illumination conditions.

All texture data needed for direct illumination evaluation is available for both the front and the back sides of a leaf, resulting in a total of eight textures per leaf. Of course, front and back side versions of a texture set may be combined into one texture, and texture coordinates altered in the vertex shader depending on which side is visible to the viewer to save texture stages. A texture set for the front side of a leaf is displayed in Figure 3.4.

Fig. 3.4: From left to right: Albedo map, normal map, HL2 map, translucency color map. Note that the albedo map additionally has an alpha-channel, with the areas displayed black being transparent. Image courtesy of Ralf Habel [29].

3.3.1 Normal Mapping

Normal mapping is the standard approach to render mesostructure in real-time rendering. A texture is used that stores normal information to distort the normal of a surface on a per-pixel basis. Typically, this information is stored in *tangent space*, which is constructed from the the surface normal N and the tangent T and binormal B. The tangent and binormal point into the direction of the largest increase of u and v texture coordinate values of the normal map. B, T and N are orthogonalized and normalized to ensure that the tangent space is an orthonormal basis. At runtime, the normal used for lighting calculations is looked up from the normal map instead of taking the interpolated surface normal directly. Therefore the light vector has to be transformed into tangent space as well, which can be done in the vertex shader.

As a result the surface appears to have detailed geometrical mesostructure on it, instead of being only one flat polygon. When rendering, these per-pixel illumination changes are especially apparant for specular highlights.

Typically, there are two ways to generate normal maps:

- The first one is by building a high-polygon mesh and a low-polygon representation of the same object. The low-polygon version is used for rendering at runtime. The normal map is generated by first calculating unique texture coordinates for all polygons of the model to store normal map data, and then transferring normal information of the high-polygon version to the low-polygon version for each texel in the normal map. This technique was used to obtain the leaf normal maps since high-polygon geometry is already available from the 3D scan.

- The second approach is to calculate depth information from the albedo map by using image-processing techniques and then build the normal map from the depth derivatives. There is a number of tools for this purpose like the *NVIDIA Photoshop Normalmap Plugin* [51] or *CrazyBump* [15]. This method was used to generate normal and displacement maps for the branch textures.

Parallax Mapping

A common addition to normal mapping is parallax mapping [36] which needs an additional height map. Parallax mapping distorts the pixel lookup of albedo and normal map according to the viewing angle towards the surface in tangent space, and taking the local texel's height into account for how far

to displace the lookup. This results in an additional three-dimensional depth perception of the surface, since bumps appear to stand out of the surface towards the viewer.

Depth information is readily available from the 3D scan, but parallax mapping proved to have no additional perceived effect on the appearance of the leaves. So parallax mapping is only used to render the branches. Most types of tree bark show considerable height differences and a generally rough surface which makes this technique very useful to enhance surface detail.

3.3.2 Shadow Mapping

The influence of direct sunlight plays a very important role in the calculation of diffuse and specular reflectance as well as translucency. Sunlight casts hard shadows, so an accurate per-pixel approach to test sunlight visibility has to be applied to achieve convincing results. Shadow mapping is the most common real-time algorithm for per-pixel shadow testing in outdoor environments.

Shadow mapping [58] is a texture-based shadowing approach. First, the scene is rendered from the point of view of the light source. Typically orthogonal projection is used for directional light sources like the sun. The distance of each rendered pixel to the light source is stored in the shadow map. When rendering the scene from the viewer's perspective, a depth comparison is made for each fragment between its distance to the light source, and the distance which is looked up in the shadow map for the corresponding texel. The view and projection matrices already used when renderering into the shadow map are used again to transform the fragment position into shadow map space. If the distance looked up from the shadow map is smaller than the measured distance, then there has to be an occluding object between the light source and the rendered fragment. The fragment is found to be in shadow.

Shadow Volumes

Shadow Volumes [20] work by extruding the geometry of each shadow caster away from the light towards infinity (in its simplest form, generating a triangular prism from each triangle), and using the stencil buffer to test which pixel of the frame buffer is shadowed. Shadow volumes always produce pixel-perfect hard shadows due to their stencil buffer-based nature, which makes them suitable for the hard shadows casted by sunlight. However, there are limitations which make them not useful for rendering outdoor vegetation environments.

The biggest disadvantage for foliage rendering is the inability of shadow volumes to handle alpha-testing. Usually, leaves are rendered as simple quads with an alpha-channel-enabled texture mapped onto them. Only pixels with alpha-values greater than a specified threshold are rendered and assumed to cast shadows. When rendered with shadow volumes, shadows of leaves would remain in their original quad form.

Texel distribution

There are some implications to shadow mapping. Since it is a texture-based method, the resolution of the texture is of course a major limiting factor to the quality of the shadows, as can be seen in Figure 3.5. Shadow maps can only be as precise as the number of texels available. Also, only one depth value is stored for each texel in the shadow map. Bilinear interpolation of the texel values is not directly possible because the stored values are depth values and not color data. Thus a whole range of depth values from the viewer's perspective are mapped to one discrete value in the shadow map. These depth differences resulting in incorrect self-shadowing or shadow acne have to be dealt with to avoid artifacts from wrong depth comparison results. Common solutions to this problem include rendering only the back faces of objects or adding a certain depth bias which can be composed of a constant value and a value scaling linearly with the angle between the pixel's normal and the view vector [77].

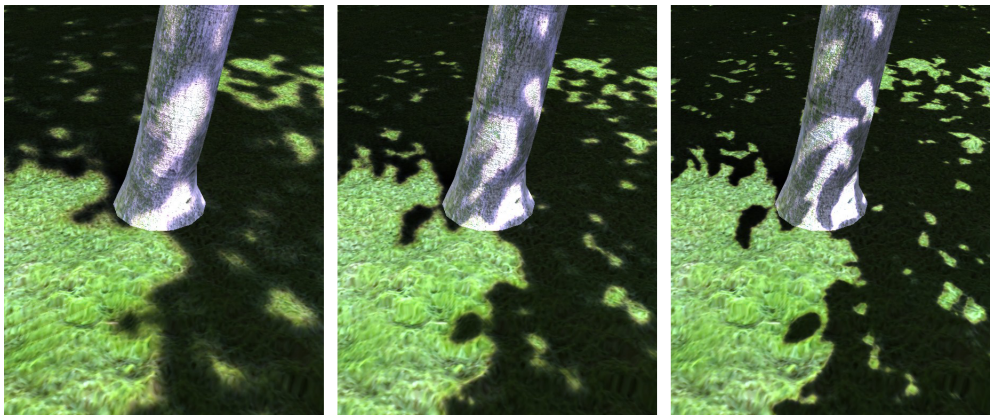

Fig. 3.5: Shadow map resolutions from left to right: 1024, 2048, 4096.

On the other side, if multiple texels are mapped to one single fragment in the view space, then *minification* artifacts are present. Finding a distribution which matches shadow map texels most closely to screen space pixels is a whole body of research on itself. Methods like *Perspective Shadow Maps* [70],

Light-Space Perspective Shadow Maps [77] and *Trapezoidal Shadow Maps* [40] warp the shadow map space to retain as many texels as possible for close-up fragments. Recently, the idea of using multiple individual shadow maps for closer and farther range in the view frustum has become more common. *Cascaded Shadow Maps* [71] and *Parallel-Split Shadow Maps* [78] are examples of this technique. All of these methods are based on finding a tight bounding volume around all shadow-casting objects of relevance (which are seen both from the light source and from the viewer's point of view) and thus *focusing* the shadow map for optimal usage of the texel space available.

A focusing technique proves to be useful especially for larger scenes if the hard shadows of individual leaves should be captured. Only uniform texel distribution was implemented in the demo application for this thesis because most of the development was done with only one tree at a time. However, integrating different focusing methods should pose no problem because shadow map tests are completely independent of the other techniques like translucency or indirect illumination.

Filtering

Hard shadow outlines are produced due to shadow map depth values being discrete per-texel values. Either a pixel is found to be shadowed or not. Filtering the shadow map results to get smooth shadow outlines is an area of research in itself. Besides the most simple *Percentage Closer Filtering* (PCF) [58], which interpolates bilinearly between the four neighboring shadow mapping results (not the depth values themselves) there are more sophisticated algorithms which sample a larger number of shadow mapping results in an area around the current texel, sometimes in dependence of the distance of the texel to the nearest occluder [21]. Techniques like *Variance Shadow Mapping* (VSM) [18] rely on statistical methods and treat a shadow map texel not as one single depth value, but as a distribution of multiple values. The results of these different filtering methods can be seen in Figure 3.6. Multiple PCF lookups were found to produce the best results, both in regard to image quality and to rendering cost. VSM produces noticeable artifacts by eliminating small spots of light in shadowed areas. This is due to blurring of mean depth values in the shadow map and the depth variance for each texel being very large for vegetation, because a large number of individual leaves with discontinuous depth values and small empty areas between them fill the shadow map.

Fig. 3.6: Percentage closer filtering leaves the shadow map texels clearly visible (left) while multiple PCF lookups smooth the result (middle) and variance shadow mapping create hard shadow outlines but eliminates fine detail (right).

3.3.3 Cook-Torrance Specularity

Specular highlights are the most prominent features seen on leaves. Usually, only the front sides of tree leaves are highly specular while the back sides are mainly diffuse. The composition of leaf surfaces varies greatly between different species, ranging from velvet-like surfaces to leaves with a thick waxy coat that create a broad intense specularity. All these stronly varying surface characteristics have to be modeled as accurately as possible.

The Cook-Torrance specular model [14] was used to model specularity for the front sides of leaves. Normals looked up from the normal map, changing over the surface area, result in broad specularity instead of the highly concentrated round highlights which are usually seen when using the "standard" specularity model of real-time rendering, the Phong-Blinn specular model [8]. Also, due to a Fresnel term specularity of leaves is mostly apparant at grazing angles, which is an effect captured well by the Cook-Torrance model.

The model is based on two physical properties, surface roughness and a refracton index. Since these are physical properties, their results are strongly non-linear and the values have to be chosen carefully. The refraction index n ranges from 1.2 to 1.7, and the roughness ρ ranges from 0.078 to 0.5 for different types of leaves. The numbers were taken from values measured by Bousquet [10] and are assumed to be constant over the whole surface of a leaf, although they could of course by varied by parameter texture maps.

Figure 3.7 illustrates the difference between the standard Phong-Blinn specularity model and the Cook-Torrance specularity model. The benefit of the Fresnel term at grazing angles is obvious.

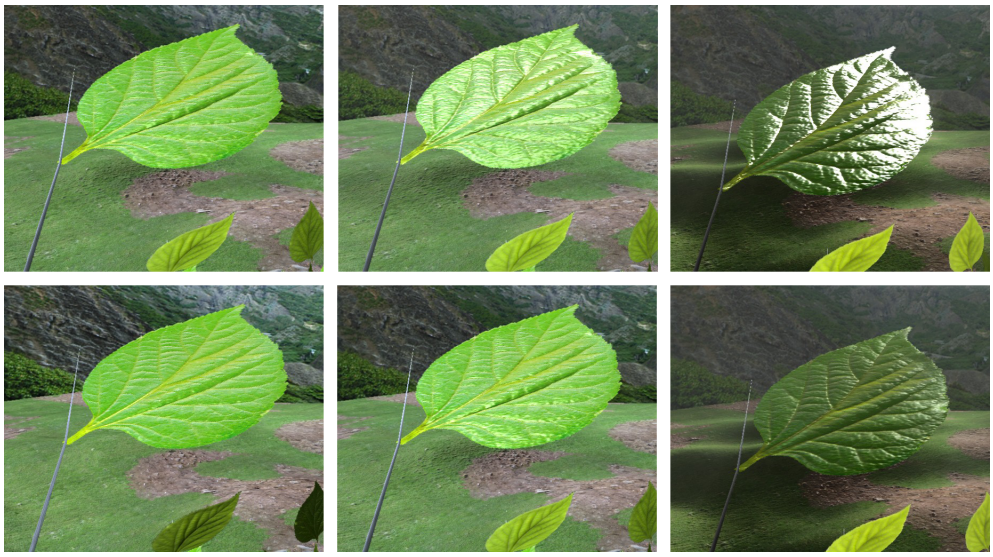

Fig. 3.7: A comparison between the Cook-Torrance specular model (top) and the Phong-Blinn specular model (bottom). The angle of incoming light changes from straight down to a low angle. The difference in specularity becomes apparant as the light is reflected at a grazing angle into the eye.

3.4 Indirect Illumination

Aside from direct illlumination coming from the sun, the hemispherical sky also has a strong impact. Additionally, a portion of the incoming light hitting each surface point is bounced off into the hemisphere above it. If this light hits another surface point in the scene it adds to the indirect illumination. This diffuse inter-reflection leads to surfaces facing away from the sun to also receive light instead of being completely dark. Calculating the effect of indirect illumination is usually handled by a global illumination (GI) solution like *Radiosity* [13] or *Path Tracing* [37], which combines direct and indirect illumination.

Usually, in real-time rendering only direct illumination is considered and a *constant ambient term* is chosen instead of a more complex indirect illumination term to avoid unlit parts of the scene being completely black. However, while indirect illumination always adds to the believability of the generated images, it is especially important for plants: Additionally to the diffuse inter-reflection there is also a large amount of light *transmitted* through leaves, which makes the translucency an integral part of the indirect illumination solution. The available light decreases continuously towards the center of the treetop as more and more environment light is filtered out by layers of leaves. Indirect illumination enhances the sense of perceived *depth* to

the image, because the different layers of leaves stand out more clearly to the viewer. Without sophisticated indirect illumination, there are mainly blotches of bright and darkened leaves, as can be seen in figure 3.8.

Fig. 3.8: A constant ambient term (left) and per-vertex ambient occlusion (right).

3.4.1 Ambient Occlusion

A simple but efficient algorithm to simulate diffuse illumination is *ambient occlusion* (AO) [55]. The amount of incoming light from the environment is assumed to be constant over the whole hemisphere above a surface point. The amount of exposure to incoming light directly determines the surface brightness. It can be calculated by doing multiple ray casting tests into different directions over the hemisphere of the surface point. The sample results which do not hit any occluding geometry are assumed to let outside light from the environment come in. The ray cast results are summed up and normalized to a user-defined scale and used to attenuate the surface point's albedo. This results in isotropic direct illumination.

Indirect illumination can be evaluated with AO as well by using multi-bounce ambient occlusion, which evaluates the AO again for each ray hitting an object recursively until a specified recursion depth is reached. Also, translucency can be taken into account by not eliminating the influence of a ray hitting an object completely, but instead modulating its influence and continuing the AO evaluation with multi-bounce ambient occlusion. Because AO does not depend on the direction or intensity of incoming light, it can be used as a simple approximation of indirect illumination while direct illumination is evaluated using other dynamic real-time rendering algoritms.

Often a simple ambient occlusion solution suffices to enhance the perception of depth and light distribution in a scene, and more complex methods like precomputed radiance transfer are not needed to push the believability of a scene any further. This is due to changes in indirect illumination being of low frequency and most observers not recognizing approximated or simplified indirect illumination in contrast to clearly noticeable artifacts in direct illumination. Due to the complexity of vegetation an AO term is sufficient for a concinving skylight.

Per-Vertex

It is not necessary to have the same density of sample points which would be needed for direct illumination. Vegetation objects tend to have a very dense polygonal mesh anyway, so a per-vertex solution with intensity values interpolated linearly over the polygon faces suffices.

Calculation

Leaves are assumed to be translucent and transmit a certain portion of the environmental light through their surfaces, with different amounts of translucency for different wavelengths. The HL2 map, which was calculated primarily to evaluate translucency, is taken into account here to obtain meaningful translucency information for each ray intersection point when calculating ambient occlusion. Ambient occlusion colors are also not simple intensity values but RGB color values. The average translucency color map is used to change the amount of transmitted light for different color channels.

This multi-bounce AO solution leads to slowly attenuating light intensities through multiple layers of leaves. The visible light turns greener and darker with each passed layer. Figure 3.9 illustrates the ambient occlusion values, as they turn from white to a dark green.

3.5 Translucency

Rendering detailed translucency is crucial for believable leaf rendering. While specular reflectance remains the most important factor to rendering the sun-facing side of a leaf, translucency is the dominant factor for the back-lit side. The algorithm presented in this thesis takes varying leaf thickness, self shadowing due to risps and bulges on the leaf surface as well as subsurface scattering inside the leaf into account. A per-pixel translucency transport function is precomputed by transferring the intensity of incoming and outgoing light over the hemisphere above the surface into the “Half Life 2” (HL2)

Fig. 3.9: The ambient occlusion values are calculated per vertex. The colors turn from white to a dark green inside the treetop.

basis. The data is stored as a texture, the so-called *HL2 map*. This allows efficient and fast rendering of translucency in real-time applications because the HL2 basis is constructed from only three orthogonal basis vectors.

The physically based subsurface simulation for thin slabs by Donner and Jensen [19] is used as a foundation to compute the contents of the HL2 map. This allows modeling subsurface scattering effects as they are found in leaves. While still using the simplification of assuming a homogeneous material, this already proves to be an advantage over other methods which basically use a Lambertian translucency model, not taking subsurface scattering effects and microstructure into account [74]. The differences can be seen in Figure 3.10. The amount of transmitted light changes with the structure of the leaf, revealing risps and veins.

Using the HL2 map not for full direct and indirect illumination evaluation but only for translucency evaluation allows keeping the data local to a leaf. The HL2 map can be instanced for every leaf in the tree, not re-aquiring any per-leaf data. Additionally, evaluation of the HL2 map can be easily combined with normal mapping because the HL2 space is equivalent to the

Fig. 3.10: A comparison between a Lambertian translucency model (left) and the per-pixel perturbed translucency (right).

tangent space. This eliminates the need to transform the incoming light vector into a different basis first, as is the case with spherical harmonics or Wavelets [39].

3.5.1 The Half Life 2 Basis

The "Half Life 2" basis was first introduced by the *Source Engine* by Valve Corporation in 2004 [42]. It allows a fast and efficient combination of normal mapping and precalculated light maps. Basically, instead of using one light map for one surface normal, 3 light maps are calculated and their results are weighted according to a normal from a normal map.

The HL2 basis is constructed from three basis vectors which are distributed uniformly over the hemisphere above a surface point. The HL2 basis vectors are orthogonal, resulting in the fact that the angle between the tangent plane and each vector is identical. The three basis vectors therefore are:

$$\vec{H_1} = \left(-\frac{1}{\sqrt{6}}, -\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{3}}\right)$$
$$\vec{H_2} = \left(-\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{3}}\right)$$
$$\vec{H_3} = \left(\sqrt{\frac{2}{3}}, 0, \frac{1}{\sqrt{3}}\right)$$

Figure 3.11 illustrates the three basis vectors on a hemisphere.

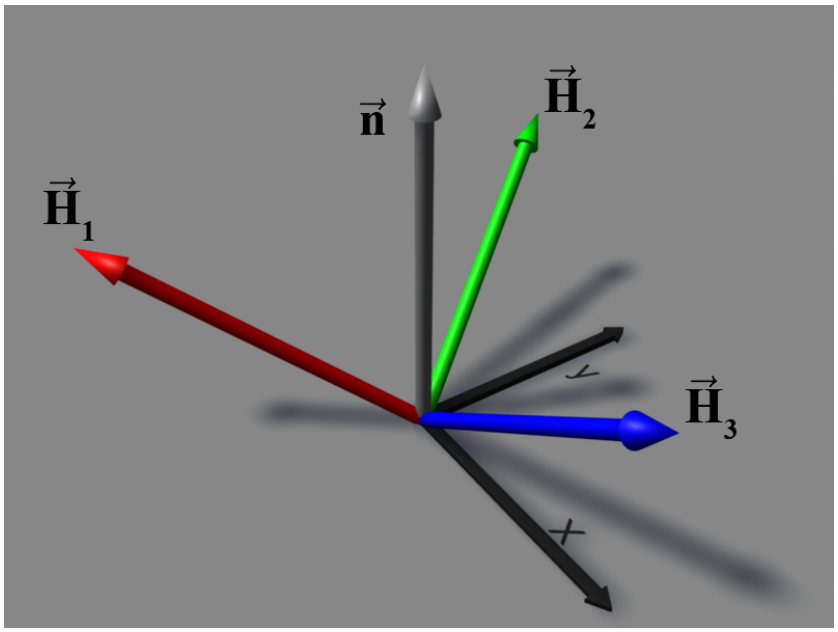

Fig. 3.11: The three basis vectors that construct the HL2 basis. n is the surface normal. x and y are the tangent and binormal vectors. Image courtesy of Ralf Habel [29].

The basis vectors are constructed in tangent space. This allows evaluating the influence of each coefficient by a simple dot product between the basis vector and the tangent space normal, which is taken directly from the normal map. No transformation between spaces is needed. This is a vast improvement over other bases, like spherical harmonics [65] or Wavelet bases [39], which require the incoming light to be transformed into the respective basis coefficients.

In the case of the *Source Engine*, instead of calculating one single light map using the world space surface normal, one light map is generated for each basis vector transformed into world space. This results in three lightmaps. At runtime, the results of these three lightmaps are combined by simply adding

the weighted color values. The weights are calculated by the dot product between the normal stored in the normal map, and the basis vectors. Dot product results are allowed to be negative as well, since the contribution of each basis vector is included over the whole hemisphere. Figure 3.12 illustrates the areas of positive and negative weights of the three coefficients.

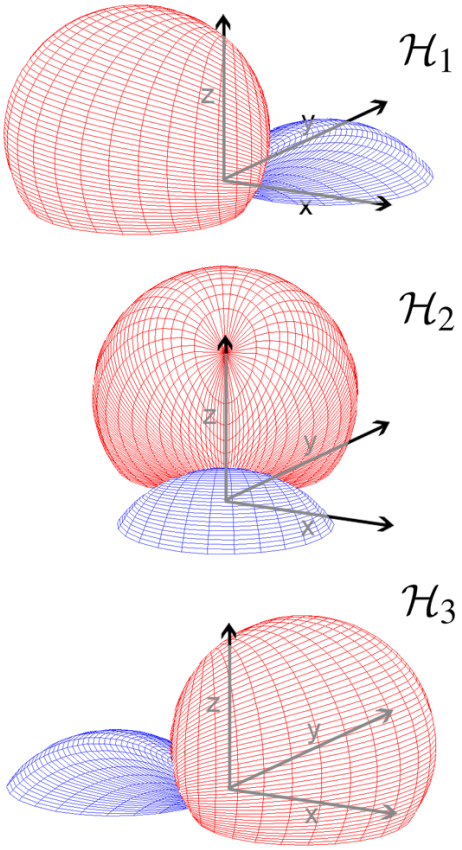

Fig. 3.12: Red indicates positive values for the coefficients, blue indicates negative values. The values of all three coefficients sum up to 1. Image courtesy of Ralf Habel [29].

Of course, the results are of relatively low frequency compared to solutions using a larger number of bases like higher order spherical harmonics because only three coefficients are available. However, if HL2 maps are not used for high-frequency information, they prove to be fast and efficient. Finally, calculating the result of information stored in the HL2 basis comes up to only 2 texture lookups and 3 added and weighted dot products. (In the case of the *Source Engine*, RGB color values are used for lightmapping, so there are effectively 3 RGB lightmaps, making the total amount of texture lookups rise

to 4) This makes the HL2 basis a very suitable choice for real-time rendering of low frequency precomputed data over the hemisphere above a surface.

3.5.2 Subsurface Scattering

A full 8D Bi-directional Scattering Surface Reflectance Distribution Function (BSSRDF) is required to evaluate a full subsurface scattering solution [49], modeling the amount of light exiting the medium at any given point. However, in contrast to most subsurface scattering materials, which can be modeled as semi-infinite slabs (for example skin rendering), leaves are a very thin and highly absorbing medium and a certain amount of light rays exit on the other side of the leaf. This keeps the range of light diffusion down to a few millimeters at maximum. This helps because a BSSRDF requires a locally flat space. With a thin medium as a leaf, this can be assumed to be the case even for bent leaves, as the space of the effective scale of the BSSRDF still remains *locally* flat.

A full BSSRDF takes variations in surface thickness into account, as well as self-shadowing of the surface and variations in the reflectance properties. Changing reflectance properties are handled by the average translucency color map, which was taken from a photograph of a real leaf. Self-shadowing is dealt with by horizon maps [41], which are precalculated offline from the measured normal maps and thickness map.

In context of the illumination model presented in this thesis, the subsurface scattering solution is assumed to be *local* only. Global lighting effects are not taken into account because the amount of incoming light from direct illumination is already determined by shadow mapping, and the light source is reduced to a directional light only instead of a full hemispheric light. The amount of incoming light only modifies the resulting radiance but does not take any noticeable effect on the subsurface scattering results itself. This is due to shadowing happening on a much larger scale than subsurface scattering.

3.5.3 Calculating Transmittance

Donner and Jensen introduced a model for an efficient approximation of subsurface scattering in a thin homogeneous slab [19]. As in the calculation for semi-infinite homogeneous slabs, the BSSRDF is approximated by a diffusion dipole, which is modeled by two virtual point light sources, positioned inside the participating medium as well as outside. In a thin slab a boundary condition is added for light exiting on *either* side of the object to never return. Multiple dipoles are needed to match these conditions. As a simplification

a leaf is assumed to consist of only one layer of homogeneous material. One problem with Donner and Jensen's method which still remains is taking into account arbitrary boundary conditions, which are needed to approximate the risp sides of leaves closely. In that case Monte Carlo simulation [4] proves to be superior. However, the error compared to a Monte Carlo solver stays in a margin of 3% for most of the surface area with exception of the risps [29].

All light that is not reflected is assumed to be transmitted through the medium. Figure 3.13 shows that the transmittance decreases exponentially with increasing thickness. The influence of incoming light to the exiting point on the opposite side of the medium decreases in a smooth Gauss curve-like falloff depending on the distance. This helps when calculating the HL2 map, because not every texel has to be compared to the full leaf surface area and a cutoff can be applied.

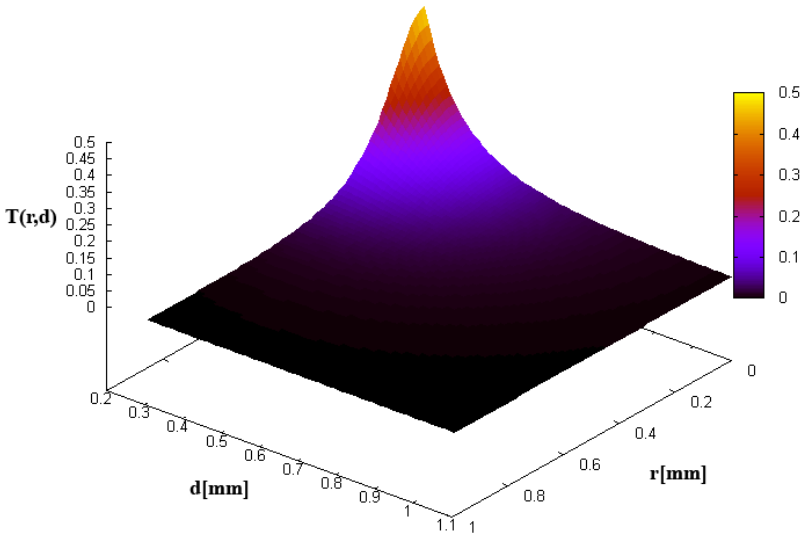

Fig. 3.13: The amount of transmitted light decreases exponentially with increasing thickness, and slowly in a smooth curve depending on distance. Image courtesy of Ralf Habel [29].

3.5.4 Calculating the HL2 map

The continuous transmittance function is discretized to an area integral to match one texel of the HL2 map, which has the same resolution as all the other leaf textures. The amount of transmitted light is sampled for many directions on the hemisphere of incoming light and then projected into the

3 directions of the 3 HL2 basis vectors. (128 samples were taken by the tool used in this thesis.) Instead of full trichromatic RGB values, only one dominant wavelength of 510nm is evaluated, which is green light. Therefore only one texture is needed to store the 3 HL2 coefficients. Although only luminance is stored, it can be quickly expanded to the RGB spectrum by modulating it with the color values stored in the translucency map.

Because the HL2 basis only consists of 3 coefficients, the results are of relatively low frequency if compared to solutions with a larger number of bases like spherical harmonics. Thus a slight blurring of the transmittance results can be noticed. Fortunately this matches the blurring properties of subsurface scattering.

All luminance values stored in the HL2 map are physical units, so a tone mapper may be required to adjust them to meaningful values for output. Due to their nature these values could also be used for a physically based ray tracer to integrate the transmittance over the whole hemisphere for global illumination.

Calculation of the translucency information at a resolution of 1024*1024 takes about 45 minutes for one leaf data set. The resulting HL2 map calculated can be seen in Figure 3.4. Details on the calculation of the HL2 map and the projection of the subsurface scattering results into the HL2 basis can be found in [29].

3.5.5 Discussion

One big advantage of choosing the HL2 basis over other precomputed light transfer solutions like spherical harmonics is that the 3 basis vectors correspond to the tangent space, which is needed anyway for normal mapping in the pixel shader. This also makes rotating the basis trivial, which is important for animation. Besides, evaluation of the HL2 basis is very cheap because only 3 coefficients are involved. This results in the calculation of the translucency in the pixel shader being signifcantly less costly than other parts of the illumination like the Cook-Torrance specular model.

Only storing transmittance coefficients in the HL2 map also makes it *local* to a leaf, so the same HL2 map can be instanced for all leaves, just as the other textures are. This makes the HL2 map not only a fast but also a very memory-efficient technique compared to full PRT solutions, which requires unique coefficient information for every texel on a plant [74]. Of course this is also important when animating the plant, where static precomputed lighting techniques are not applicable.

Chapter 4

State of the Art on Plant Animation

Trees are very complex systems of a large number of interconnected branches, which all have to react to wind. Large branches usually move at a lower frequency than small twigs or leaves. Ideally, also leaves react to wind individually. In addition to the high geometric complexity posed by this problem, individual elements usually do not stay rigid but bend smoothly when reacting to incoming wind. Wind itself is not only a simple directional force but a three-dimensional turbulent air flow which gets influenced as it hits the tree.

Modeling of the animation of plants therefore consists of different components:

- A *structural model* of the plant itself is needed. The physical properties of thinning branches and leaves should be taken into account. Usually a hierarchical joint system is used, which allows propagating branch motion down to sub-branches and leaves. Structural representations of plants are described in Section 4.1. However, there is also a completely different approach which is solely based on vertex displacement and does not have any connectivity information or hierarchical structure of the plant at all.

- *Wind* can be modeled either as a three-dimensional velocity field or as a stochastic process. Simulation allows feeding branch motion back into the system to create turbulent wind movement automatically, while stochastic approaches have to treat free-flowing directional wind and turbulent wind moving through the treetop separately. Since wind is invisible to the naked eye and the interaction of wind and tree is of a very complex nature, this component is well suited for simplification.

- An *animation scheme* has to be found to modify branch motion and wind for each frame. Animation can be performed by either integrating the equations of motion in the structural model of the tree and the wind model, or by heuristic approaches that try to emulate the characteristics

of branch swaying and bending as closely as possible. More than one structural element is needed in a single branch to perform a realistic bending motion.

The different methods described in this chapter employ various combinations of these 3 components with different goals in mind: Most algorithms only strife for a convincing *look* of the resulting animation and have no aim to correctly simulate the complex behavior of tree and wind. Especially wind is often modeled as a stochastic process without complex spatio-temporal correlations. The different methods are described in Section 4.2.

Most approaches deal with the animation of only one highly detailed plant, but there is also need for approaches simple enough to animate a large number of plants at a high framerate and still leave enough resources for advanced shading techniques or additional CPU computations. This totally different method is discussed in Section 4.3.

4.1 Plant Representations

Trees are usually represented as an interconnected hierarchical structure of a trunk, multiple levels of branches and leaves at the end. These components are the *structural elements* that are used to build the tree. Animation can be applied to the joints connecting them by rotating them, thus resulting in a correlated hierarchical swaying motion. Branches should also bend to the influence of incoming wind to create a convincing result. Most methods regard a branch as one rigid structural element, which does not allow bending. A branch has to be separated into multiple structural elements to achieve bending. Typically bending is realized by applying a *structural mechanics* model to a branch, which deals with the deflection and internal forces and stresses within the branch. A feasible model to use for branches is the comparatively simple Euler-Bernoulli model [72].

4.1.1 Structural Elements

The dominant model used to represent the hierarchical structure of a tree is a rigid skeletal joint system [1, 52, 62, 69]. Skeletal systems are a common method to animate dynamic objects or characters in computer graphics. An object is built from a number of connected joints (with the connections typically called “bones” in character animation). An example of a hierarchical structure used to represent a tree can be seen in Figure 4.1.

Animation of the object is performed by transforming the frames of these individual joints, usually by rotation. Because the joint system is a hier-

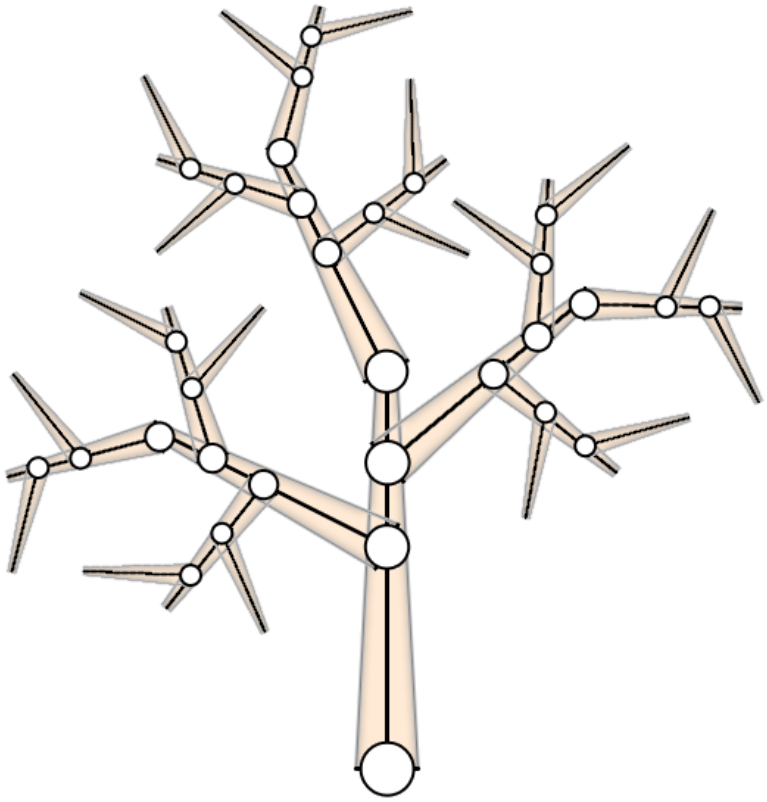

Fig. 4.1: A hierarchical structure of joints representing the structural elements of a tree. Each branch is modeled as one structural element.

archical tree structure, the whole child structure of a joint is rotated with the parent. Each vertex is assigned one joint in the case of rigid skinning or multiple joints in the case of smooth skinning. Animation is applied by transforming the vertex position and normal with the joint's frame. If smooth skinning should be applied, then multiple joint transformation matrices are modulated in regard to weights defined per vertex which sum up to 1.

When rendering the object, the joint transform matrices are usually uploaded to the GPU and the process of skinning is performed entirely on the GPU. (Of course this is only possible if the number of shader constants needed to store the joint transform matrices does not exceed the maximum number of usable registers on the graphics card.) The calculation of the hierarchical transformation matrices each frame is an obvious performance limitation, though. This usually has to be done on the CPU, because hierarchy depth varies vastly per joint and transformation calculations have to be performed

in the correct order to retain the hierarchical nature of the transformation. All this makes parallelization on the GPU very difficult and inefficient.

While the calculation of the joint transformation matrices is a relatively fast process, the calculation of the rotation information itself can be of high complexity if an expensive simulation is used. The complexity of evaluating the system at runtime therefore usually scales linearly with the number of joints used. So the number of branches has to be kept low to maintain real-time framerates [60]. To achieve bending of each branch as a reaction to incoming wind, a number of joints is needed per branch, which further increases the complexity of the simulation. This disadvantage can be overcome by using a *structural mechanics* model, as is described in the next section.

4.1.2 Structural Mechanics

Structural mechanics describes the deflections, deformations, and inner stresses within structures. While there are very sophisticated models, the comparatively simple Euler-Bernoulli beam model proves sufficient to model deflections for bending of branches [72]. Branches are assumed to be cylinders of homogeneous physical properties, thus simplifying calculations. If the cylinder is thinning out at the free end, then a strong non-linear deflection can be observed creating the bending behavior of branches seen in nature. The Euler-Bernoulli Beam model is also the basis to calculate bending in the approach intoduced in this thesis and is discussed in detail in the next chapter in Section 5.2. The benefit of smooth bending compared to a rigid branch model can be seen in Figure 4.2.

To model a highly detailed tree with over 1500 branches and ten-thousands of leaves with smooth bending, more than 30000 joints would be required. Using a full evaluation of the hierarchical structure each frame is obviously very expensive to calculate. So most methods rely only on rigid structural elements like the one proposed by Akagi [1], Shinya [62] or Ota [52], ignoring bending alltogether. Bending is often incorporated in methods which rely on precomputation [69, 33].

A major limitation of using a unified hierarchical structure is the inefficiency of animating individual leaves. If a tree has thousands of leaves, then the same number of joints would be needed to animate them, even with the simplification of the leaves being treated as rigid objects. If bending should be included, the number of joints has to be increased even further.

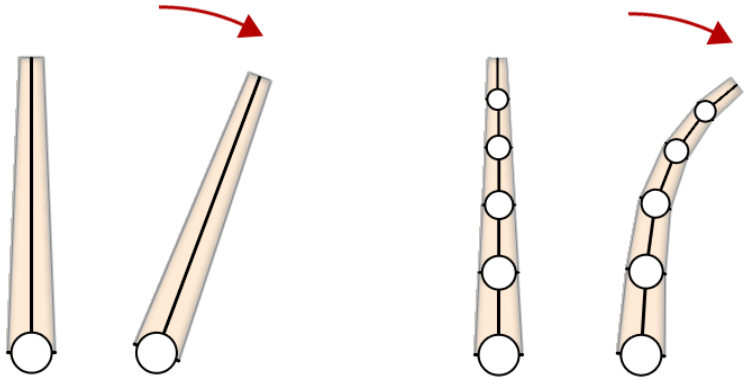

Fig. 4.2: Smooth bending becomes available if a structural mechanics model is applied. The same rotation is performed by only one structural element (left) and five elements(right). The more elements are used, the closer the model matches reality, although each cylindrical part of the branch is still bound to only one joint.

4.2 Applying Animation

Wind has to by updated every frame in order to keep the animation running. Evaluating wind flow as a three-dimensional fluid simulation or as a velocity field is very expensive to compute [1]. Therefore, most methods use stochastic wind models instead. Wind is represented as a noise function generated in the frequency domain from the power spectrum of branch motion with respect to a dominant resonance frequency. By applying FFT (Fast Fourier Transform), the signal can be transferred into time domain and sampled each frame to obtain animation information. This only simulates turbulent wind, as no clear wind direction is encoded into the power spectrum. However, using a stochastic wind model allows considering only the noise input and the resulting motion vectors for the current frame without the need to integrate from the last frame forward. This greatly simplifies calculations, although aspects like intersections of branches or reactions to exterior influences cannot be included. Strong directional wind has to be treated separately, too.

4.2.1 Simulation

Akagi et al. [1] use a full three-dimensional fluid dynamics simulation, solving the Navier-Stokes equation for incompressible fluids to model the influence of wind on the joints. The rotation of structural elements in the tree is calculated by explicit integration of the branch motion and the results are fed back into the fluid dynamics simulation. This allows evaluating directional

wind as well as turbulent wind in the same way, as well as reacting to possible environmental influences. Another benefit of a full simulation is that intersections of branches can be detected and avoided.

Of course this is very expensive to calculate in real time. A boundary condition map representing resistance from the tree in the 3D field is generated to speed up calculations. Still calculations for each frame of animation are too expensive to be feasible for a real-time system running at 60 frames per second, even though a comparatively simple Euler integration is used. A tree animated by Akagi's method can be seen in Figure 4.3. Sakaguchi and Ohya [60] simplify integration by ignoring the effect of gravity. Joints are animated as angular springs to create an oscillating swaying motion.

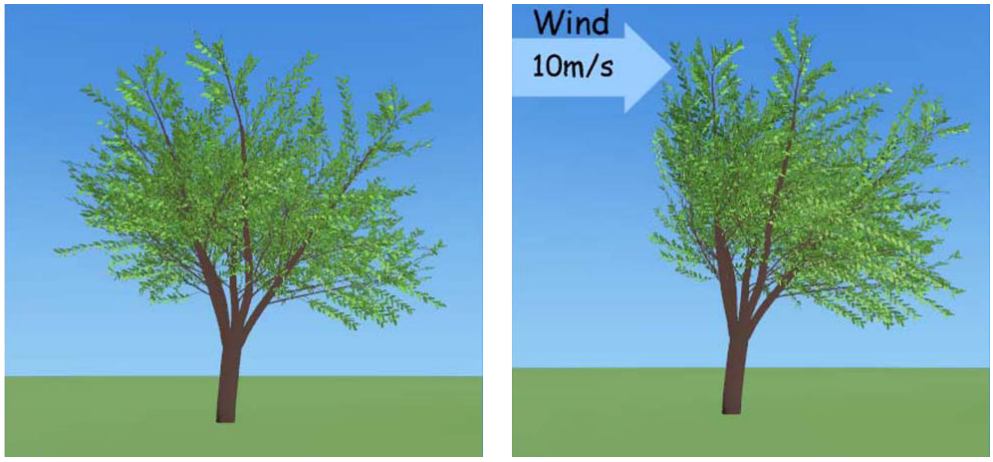

Fig. 4.3: A tree animated by Akagi's simulation.

4.2.2 Stochastic Approaches

Shinya and Fournier [62] and Zhang et al. [78] use a stochastic wind model instead of solving the Navier-Stokes equation. Wind is modeled as a three-dimensional velocity field with 3D wind vectors for each grid point. A stationary Gaussian process is used to evaluate wind velocity vector fluctuations. Wind is generated as a power spectrum in the frequency domain and FFT is used to transform it back into time domain. This approach is also taken by Yung et al. [11], who use spectral methods to model various natural phenomena. They consider branches as harmonic oscillators, swaying at their resonance frequency.

Jos Stam [69] uses modal analysis to find the natural modal shapes of branches moving in wind. Branch motion is simulated in the frequency do-

main, which allows applying spectral methods and avoiding an explicit integration of motion. Branches are represented as flexible structures, vibrating freely at different modal frequencies. Multiple modal shapes are precomputed and simply superimposed at runtime to obtain the final appearance of a branch reacting to an arbitrary wind force composed from those modal shapes. This process removes high-frequency motion of small branches from the simulation but allows fast evaluation due to preprocessing of the modal shapes. Bending of branches is also included in the preprocessing step. Figure 4.4 shows a tree animated by this method.

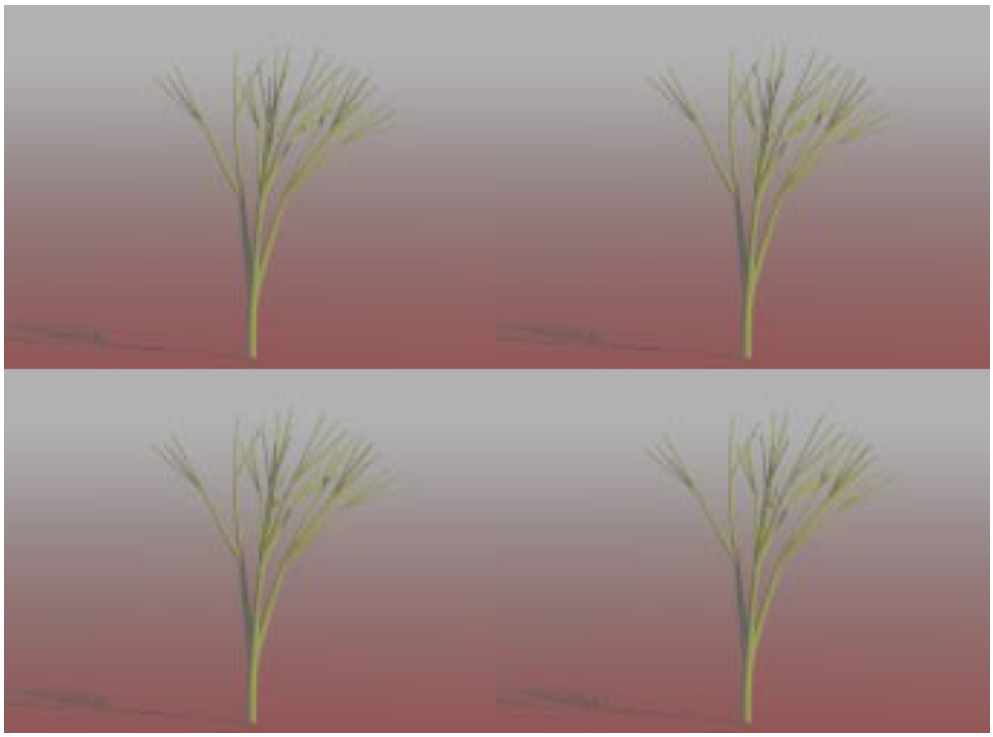

Fig. 4.4: A tree animated by Stam's approach. Bending along the branches can be observed in the different states of animation.

Haevre et al. [76] also rely on precalculation. They use a technique similar to motion graphs to precalculate a set of motion samples. Motion graphs essentially are directed graphs of animation states with predefined transitions depending on varying input. Re-sequencing these samples at runtime gives a controllable and directable goal-based motion, but of course the variety of results is limited to the number of precalculated shapes.

Diener et al. [17] also base their work on modal analysis. Instead of fully evaluating the modal shapes each frame – like Stam does – they pre-calculate a set of modes offline for a given wind basis. The modal shapes are updated

each frame on the GPU in a pixel shader with one set of modes per tree instance. This allows changing the wind direction in real time for a large number of trees in a scene. A LOD approach is used to reduce the number of modes for farther-away trees to only the largest deflections. So, only branches are animated and leaves are simply dragged along by them. However, as with Stam's technique, high frequency oscillations cannot be fully captured with the precomputed modal shapes. The method also shares the limitations of skeletal tree animation methods, and does not animate leaves. Results from Diener's method can be seen in Figure 4.5.

Fig. 4.5: A tree and a forest scene animated by Diener et al. Their approach builds on modal analysis and allows animating a large number of trees on the GPU only.

4.2.3 Heuristic Models

Heuristic animation systems do not try to *simulate* the motion of branches as correctly as possible, but instead try to emulate the *appearance* of movement using noise functions. Ota et al. [52] use noise functions to drive the rotation of branches. A spring model is used to make branches sway when reacting to incoming wind. This leads to a correlated motion behavior of similar branches, but the resonance frequencies of the branches are not considered. Ota's approach only models turbulent wind. Strong directional wind is not taken into account. Results from Ota's method can be seen in Figure 4.6.

4.3 A Vertex-based Method

A completely different approach was taken by Sousa [67] in 2007. The system was developed primarily for the use in the computer game "Crysis" [26], so the aim was to provide a very fast and efficient animation solution. Physical foundations or stochastic observations of real vegetation were not a priority. This approach is particularly interesting because it was created with the aim to be extremely efficient on resources and memory. Hundreds of plants

Fig. 4.6: A tree animated by Ota's method. Only a small number of branches is present because animation is bound to rotation of individual joints on the CPU.

can be visible at once so even the most simple skeletal animation system proves to be too expensive to be evaluated per plant. A simple vertex-based method was found that can be executed solely in the vertex shader without any additional CPU calculations.

4.3.1 Vertex Displacement

The complexity of the animation system is very limited because information on how to animate a vertex is local to it and no information about adjacent vertices or a full hierarchy structure is available. In Sousa's method vertices are only displaced along the XZ plane depending on wind strength, wind direction and a weighting factor, and the free ends of branches or large leaves are displaced along the Y axis (assuming Y points upwards) [67]. Only vertex positions are modifed. Normals and tangents can not be recalculated because no vertex adjacency information is available and no coordinate frame can be constructed to rotate them.

The *amplitude* of this displacement is specified as a modulation factor set on a per-vertex level manually by an artist. The animation *frequency* is constant for a plant and is only modulated by a global wind strength. While having an artist fine-tuning the parameters for each branch and each leaf helps creating a convincing animation from such simple input, it is also a very tedious and time-consuming process.

4.3.2 Animation

Sousa has no detailed information about turbulent wind at all. Wind is only treated as an intensity curve with lookup values changing every frame. The curve is constructed from a constant set of multiple triangular waves which are evaluated and their results smoothed out in the vertex shader. The curves are defined by hand and are guaranteed to repeat periodically. To avoid having all branches and leaves in a plant move in the same fashion,

an additional offset value is introduced per vertex to displace the lookup in the curve. This offset is also set by hand and should be uniform per branch to retain coherent motion. Additionally, a global offset is randomly assigned to each plant instance to avoid all plants in the scene performing the same animation at once. The amplitude of motion depends linearly on wind strength and is varied each frame to avoid an obvious periodic repetition of motion. Wind direction is modeled separately by shearing the whole plant into the direction of the wind vector with the amplitude varying over time to add variation.

Obviously the shear motion in XZ direction and swaying limited to the Y axis makes this model impractical to animate complex broad-leaf trees with multiple hierarchical levels of branches, but it is suitable for trees with long trunks and the treetop distributed mainly on top of it. The animation model is used predominantly for jungle trees and palm trees, for which this is the case. Figure 4.7 shows a palm tree animated by this method.

Fig. 4.7: A tree animated by Sousa's approach. The whole tree is sheared along the XZ plane and the large palm leaves move up and down on the Y axis.

The model was also expanded to work together with a skeletal system of mass springs which were applied to the larger first-level branches of bushes near the viewer. This allows the branches of bushes to be rotated away from characters moving through the environment and bounce back after the character leaves. Still, the animation system is fast enough for real-time animation of a large number of vegetation objects while leaving enough resources for advanced shading effects, real-time illumination and game logic calculations on the CPU.

One big advantage of this method is that it scales linearly with the number of vertices and not with the number of branches in the mesh (which are organized hierarchically and therefore are more expensive to evaluate). This is why a *vertex-based* animation system was deemed the most useful method to animate a complex highly detailed tree with thousands of branches and was

adopted by the algorithm described in this thesis. Also a method evaluated in the vertex shader is obviously well suited to animate a large number of individual leaves. As the number of vertices can be scaled with LOD (level-of-detail) techniques, it is possible to use the same animation complexity for a highly detailed tree with thousands of individual branches and for a far-away simplified billboard-cloud version which may have only one or two levels of hierarchy represented at all. Of course, many additions and extensive precalculations have been added to increase the complexity of the animation system. The algorithm is described in the following chapter.

Chapter 5

A Physically Guided Real-Time Vegetation Animation Algorithm

This chapter describes an algorithm designed to animate vegetation in a real-time environment based on physical foundations. The runtime component of the algorithm is executed in a vertex shader, therefore it can be easily integrated in any modern rendering system. No additional computations on the CPU are required. Thus, rendering cost scales linearly with the number of vertices and independently of the branch structure complexity, which works well with LOD (level-of-detail) techniques ubiquitously present in outdoor scene rendering. All precalculations of necessary data are done offline in separate tools. Tree geometry created by standard tree generation tools like *XFrog* [28] or *natFX* [6] can be processed automatically without any additional user input.

The algorithm was designed with multiple goals in mind:

- First of all, it should be independent of the *complexity* of the underlying tree structure. A local per-vertex model was chosen so the number of branches in a tree does not impact performance. High-detailed tree meshes with thousands of branches can be animated the same way as simple billboard clouds for far-away LODs.

- *Non-linear bending* of branches has to be possible. When affected by wind, branches do not stay rigid but are bent according to their physical properties.

- Since *individual leaves* can be rendered at high detail they should be animated individually, too.

- Finally it should be easy for an artist to tweak the animation at runtime without the need of tedious recalculations. A *set of meaningful parameters* is defined which allows easy fine-tuning of the appearance of the animation.

Figure 5.1 shows an animation sequence for slight and strong wind.

Fig. 5.1: An animation sequence for slight wind (top) and strong wind (bottom). Non-linear deflection is noticeable on the lower left branch.

5.1 Overview

The algorithm described in this chapter was designed to animate complex highly detailed tree meshes under real-time constraints without posing a signifcant load on the CPU or GPU side. The runtime component of the algorithm is executed entirely in the vertex shader, thus removing all load from the CPU. A *localized* per-vertex model is used to stay independent of the number of branches in the mesh and also allows animating each leaf individually. Additionally each branch is modeled as a tapered cylinder based on a structural mechanics model to enable non-linear deflection. This allows the free ends of branches to react stronger to wind than the thicker inner parts and creates a more convincing motion.

The *Euler-Bernoulli beam model* is used to calculate the deflection of branches. The standard model only calculates a deflection along one axis, effectively creating a shearing motion. Length correction has to be applied to transform the shear into a rotational bend. Section 5.2 describes the details of this model and how it is integrated into the animation algorithm.

Section 5.3 deals with *preprocessing* of the underlying data which is needed to animate a tree at runtime. Data generation is done offline so the data has to be stored and retrieved at runtime.

The stochastic *wind and animation model* is discussed in Section 5.4. Wind is stored as a set of two-dimensional noise functions in the frequency domain. Turbulent wind is modeled as a power spectrum regarding the oscillation frequencies of branches and calculated from wind and branch properties. The intensity and frequency of motion also depends on wind strength:

Slight and medium wind is enough to move small branches and leaves but only strong wind is able to bend a whole tree into one direction. Directional wind is integrated into the animation algorithm separately.

Finally, Section 5.5 describes how all of these parts work together at runtime to calculate branch motion on *a frame-by-frame basis* without the need of knowledge about previous or following animation states. A set of parameters is introduced which allows an artist to efficiently adjust the animation.

5.2 Modeling Branches as Tapered Cylinders

Commonly, branches are treated as elastic cylinders of uniform material that bend to incoming wind [1, 52]. But branches are usually not of the same thickness along their whole length and get thinner at their free ends instead. This *tapering* leads to the free ends of branches reacting stronger to incoming wind and being deflected to a larger degree. Contrary to standard models that transform the whole branch uniformly [1, 52, 60], a non-uniform bending has to be found that takes the varying local thickness along the branch into account. The Euler-Bernoulli beam model was chosen, which describes the deflection of long thin elastic cylinders of homogeneous material and can be used to calculate a displacement value for each point on the beam.

The characteristics of this model allow a simplification which is fast to evaluate in the vertex shader but still yields virtually exact results. Also, large deflections due to strong wind have to be accounted for to avoid overstretching of the beam, because the Euler-Bernoulli beam model only describes a translation along the beam normal's plane. This leads to a shear motion, so points need to be corrected in length by moving them back towards the branch base point to retain the original length.

5.2.1 The Euler-Bernoulli Beam Model

The Euler-Bernoulli beam model is a structural mechanics model describing the deflection of a beam of uniform material according to a transversal force. It is a simplification of the linear theory of elasticity that yields reasonably close results for long thin beams of a length-to-thickness ratio of at least $15 : 1$. This model is therefore well suited for branches [72]. The Euler-Bernoulli differential equation is written as:

$$F = \frac{\partial^2}{\partial x^2}(EI(x)\frac{\partial^2 u(x)}{\partial x^2}) \tag{5.1}$$

with $u(x)$ being the unknown deflection of the beam, F the transversal

force applied to the beam, I being the area moment of inertia and E the elastic modulus, which is assumed to be constant.

In this thesis a branch is treated as a linearly tapered circular beam which is defined by its length L and the radii s_1, s_2 at the base point and the free end. Figure 5.2 illustrates the beam model.

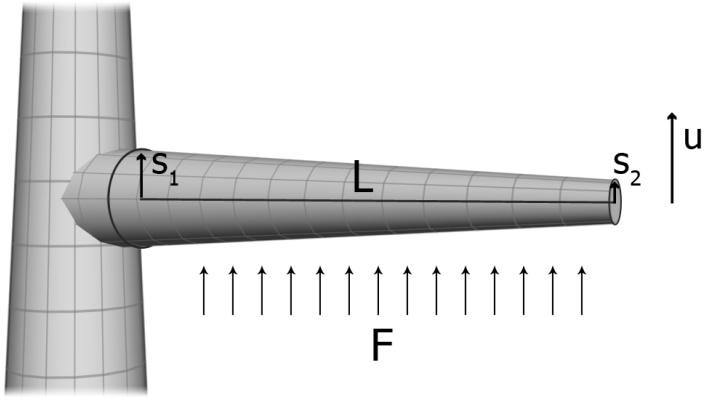

Fig. 5.2: A branch is considered to be a tapered cylinder of length L and the radii s_1 at the base point and s_2 at the free end. Image courtesy of Ralf Habel [30].

Boundary conditions can be defined because each branch is known to be fixed at its base point. Also, the length is normalized to 1 with the radii rescaled to r_1, r_2 and E' being the rescaled elastic modulus. The area moment of inertia is a function of the radius r at each given point with $I = \pi r^4/4$.

The beam is assumed to taper linearly along its length. The taper ratio α is defined as r_2/r_1. This allows an analytical solution to the equation, which still consists of a large number of terms and is therefore not well suited to be evaluated in a vertex shader directly. According to Habel et al. [30] the full equation is:

$$\begin{aligned} u(x) &= \frac{E'F}{r_1^4}\big(x(\alpha-1)(6+x(\alpha-1)(2x(\alpha-1)(3+(\alpha-3)\alpha) \\ &+ 3(4+(\alpha-2)\alpha))) - 6(1+x(\alpha-1))^2\log(1+x(\alpha-1))\big) \\ &\cdot \big(3\pi(1+x(\alpha-1))^2(\alpha-1)^4\big)^{-1}. \end{aligned} \tag{5.2}$$

An important point is that while the radius of the beam thins *linearly* along the length, the deflection is a *quartic* function of the local radius. Also, the transversal force F affects the solution linearly, resulting in a direct proportional relationship of incoming force and resulting deflection. This allows the use of wind strength to scale the deflection amplitude linearly. The base point radius influences the deflection with ${r_1}^{-4}$ [30].

So instead of evaluating the full equation in the vertex shader, a linear least squares fit is found, with the polynomal form:

$$u(x) = c_2 x^2 + c_4 x^4 \tag{5.3}$$

The taper ratio α, the rescaled elasticity modulus E' and the base point radius r_1 are baked into the coefficients c_2 and c_4. As the length is normalized, x ranges from 0 to 1.

Figure 5.3 illustrates how different taper ratios affect the deflection along the normalized length of a beam. An untapered beam with $\alpha = 1$, as is often used by structural element models to approximate branch motion [1, 52, 62], results in an almost linear deflection, which is far from what can be observed in nature. A quadric relationship between length and deflection can be observed with tapered branches. A comparison of the full Euler-Bernoulli solution to the least square fit approximations shows that the results are virtually exact for taper ratios above 0.1, with the error already below 0.2% and decreasing further with the ratio increasing [30]. A detailed discussion of fitting the Euler-Bernoulli beam model into a least squares approximation can be found in [30].

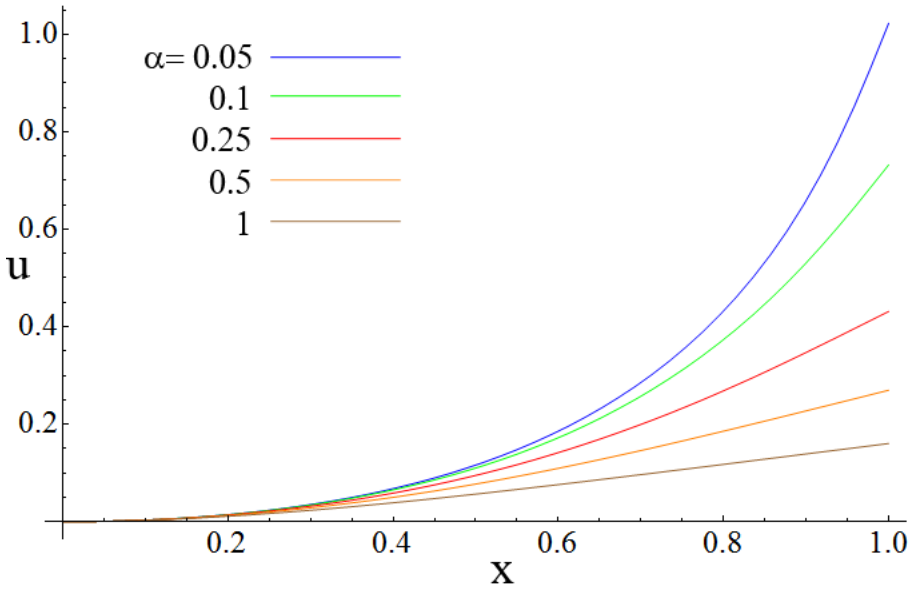

Fig. 5.3: Different taper ratios α result in different deflection intensity values $u(x)$ along the branch length x. Image courtesy of Ralf Habel [30].

5.2.2 Length Correction

The Euler-Bernoulli beam model works well for deflections with an amplitude smaller than 1/4 of the beam length. Simply shearing the points results in stretching of branches. Figure 5.4 illustrates the difference between a simple shear transform and a length-corrected rotational bend. The *distance* of a vertex to the branch base point has to be preserved. An exact solution would require solving an elliptical integral [7] as branch bending is continuous. This integral can not be formulated explicitly so it cannot be solved in the vertex shader [30].

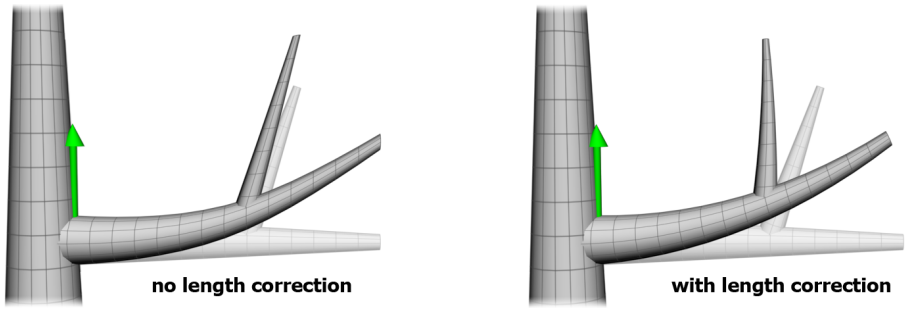

Fig. 5.4: A branch is deflected by a force along its normal. The difference between a simple shear along the branch normal (left) and a length-corrected bending motion (right) becomes clear at increasing deflection amplitudes.

By moving the deflected vertex back towards the base point along the branch tangent, the shear transform can effectively be converted into a rotation. The amplitude of the deflection is used to linearly scale the length of the vector to displace the vertex back along the tangent. Due to the fact that the deflection is essentially a quadratic function, the linear approximation still yields results close to a correct solution [30].

5.3 Generation of Animation Data

Most of the animation data is precomputed once for a tree mesh and then loaded at runtime. The main components of preprocessing are identifying branches in a tree mesh, assigning animation weights and properties to each branch and linking each vertex to a branch. Additionally, a hierarchical structure is built to store the relationships of branches. It is not possible to create correlated branch movement without such a structure, as many publications show [1, 52, 60]. The hierarchical structure of a tree mesh can be obtained without the need of user input or additional outside information like an existing joint structure.

Preprocessed data is stored either directly in the vertex stream in the case of per-vertex data, or in a separate texture in the case of per-branch data, which is then sampled as needed in the vertex shader. Data which is constant for a tree or only needed on a per-hierarchy level is stored in shader constants. Precomputation is divided in two steps: First information for the branch geometry is gathered and evaulated, and in a second step, it is propagated to the leaf geometry.

5.3.1 Hierarchical Structure

The animation algorithm described in this thesis is based on a hierarchy of individual branches. The hierarchy is used to propagate deformations from each level of hierarchy down to the next, similar to existing structural elements models [1, 60]. Because a per-vertex model is used, a branch does not have to stay rigid during animation. Motion is realized by bending a branch along $\vec{r}$ and $\vec{s}$ in local branch space. $\vec{r}$, $\vec{s}$ and the principal axis $\vec{t}$ of the branch form an orthogonal space, which is illustrated in Figure 5.5.

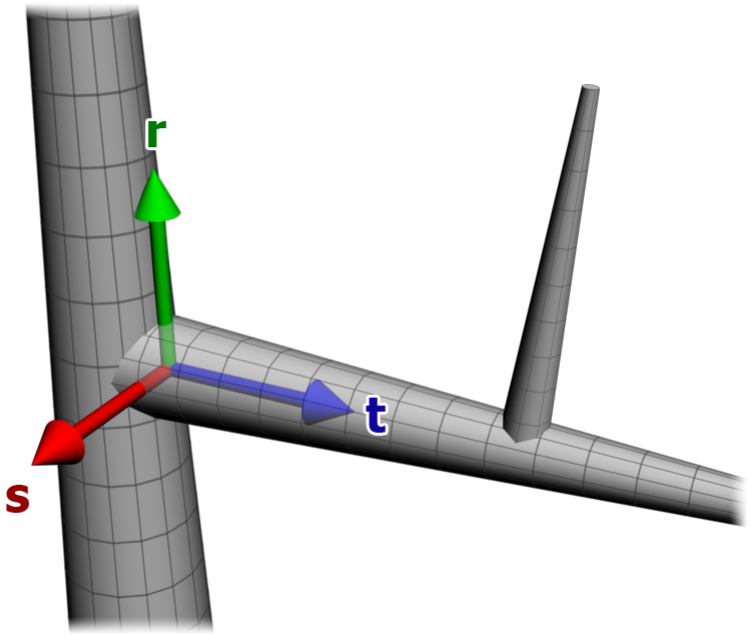

Fig. 5.5: The local orthogonal space for each branch, created from $\vec{s}$ (red), $\vec{r}$ (green) and the principal axis $\vec{t}$ along the branch (blue).

The amount of deflection is calculated by evaluating the Euler-Bernoulli beam model, which was discussed in Section 5.2. The whole branch is assumed to have homogeneous physical properties, but varying thickness results

in highly non-linear bending.

5.3.2 Branch Identification

The first step to generate animation data is identifying individual branches in the tree mesh. It is assumed that branch geometry and leaf geometry are already separated as two meshes, since they have to be rendered with different shaders anyway. This already complies to the way natFX [6] or XFrog [28] generate trees. However, if the meshes are not split yet, this can be done easily by assuming that disjointed flat quads are leaves (or billboards in a billboard cloud) while larger connected polygonal objects belong to the branch mesh.

Tree generation packages usually create trees by placing a cylindrical structure as a trunk, and then adding additional smaller cylinders rotated away from it for branches, with distribution and rescaling normally bound to the golden section. This process is recursively repeated until a desired level of sub-branches is reached and the algorithm stops. The number of segments of each cylinder is determined according to its thickness and length. Individual segments along the length can be displaced to create a more natural grown look. Additional computational tasks could be performed to ensure that no branches intersect each other, that they bend towards the light and of course that the whole tree has a balanced shape. However, the individual cylinders are usually *not* interconnected. This allows considering each enclosed polygonal shape to be one branch, as can be seen in Figure 5.6. Of course, existing joint data coming from a tree generation package could also be used to identify branches and their hierarchical structure.

5.3.3 Per Branch Data Generation

The polygonal object forming a branch is usually a cylinder. Sometimes the polygonal shape is closed at the free branch end. This facilitates finding the base, which is a circular border at the thicker end of the cylinder. The center of the base is called the "base point". The principal axis and length of the branch can then be determined from a vector from the base point to the far end of the cylinder. Similarly, thickness is calculated from the largest distance of a pair of vertices at the base.

The normal is found by normalizing a vector from the base point to any base vertex. Branch space is then calculated from orthogonalizing the principal axis $\vec{t}$, $\vec{r}$ and $\vec{s}$. Branch space is illustrated in Figure 5.5 and is needed for all hierarchical deformations.

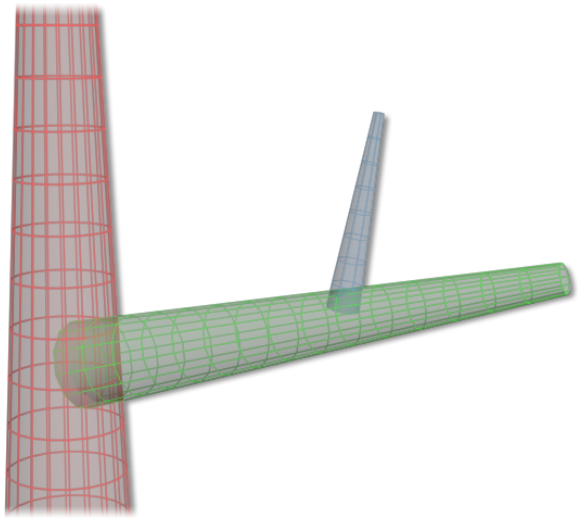

Fig. 5.6: Branches are constructed from separate tapered cylinders. Each polygonal shape is considered to be an individual branch.

Additionally a random offset value is calculated which is used for lookups in the noise texture when the branch has to be animated. This is needed so that each branch is sampled at a different position in the noise texture during animation. This is discussed in detail in Section 5.5.1. Changing the lookup position leads to different motion vectors for each branch with the overall amplitude and frequency of branch movement staying the same for all elements of one hierarchy level, as this is encoded into the noise texture itsef.

5.3.4 Hierarchy Computation

A hierarchical structure has to be calculated from a loose number of branches. The *thickest* branch is assumed to be the trunk, and assigned hierarchy level 0. Going on from there all sub-branches have to be identified, which are branches with geometry intersecting the trunk's geometry. It is safe to assume that child branches have to intersect the parent geometry because otherwise visible holes in the tree mesh would be present. If branches are ordered by their thickness, then the parent branch can be found easily by looking for a vertex whose branch is already assigned a hierarchy level and that is closest to the base point of the current branch. Obviously, if a custom tree generator is used then the data needed to build the tree geometry in the first place can be directly used to store branch information and the hierarchical structure. The process is repeated recursively until all branches have a parent and a

hierarchy level assigned to them. Of course, a maximum hierarchy level can be defined, in which case all child branches simply adopt the level of their parent instead of increasing it. The approach presented in this thesis stops at 4 levels of hierarchy, because that number fits perfectly into a 4-coordinate vector attribute in the vertex stream.

It was found that for a tree generated by Xfrog [28] with approximately 1500 branches, 4 levels of hierarchy suffice to assign a hierarchy level to all branches of the mesh. The majority of branches (about 1200) are already in the fourth level, so the geometric detail of the mesh is already very high. Also, leaves attached to these fourth-level twigs create an enclosed dense volume that forms the treetop without the need to add any additional levels of hierarchy for a more convincing animation.

The data generated for all branches including the parent relationships are encoded into a texture called the "branch data texture". This texture is unique for each tree mesh, and has to be loaded into the vertex shader. Each branch is stored in a column of the texture with branch space vectors, offset values and the parent index stored in texels beneath each other.

5.3.5 Per Vertex Data Generation

Most data is consistent per branch. But in order to apply a non-linear deflection, a value is necessary to store the deflection intensity for each vertex on the branch. The intensity can be calculated by evaluating the Euler-Bernoulli beam model for the local thickness at the vertex position.

For a start, the weight values are distributed linearly along the length of a branch. Values are normalized to the range of $[0, 1]$ with 0 at the base end and 1 at the free end. If these values were used directly as deflection modulators, a linear bending of the branch would be achieved. This behavior corresponds to the shearing of structural elements operating only on the branch level [1, 52, 60]. But a structural mechanics model is used instead, which takes the weight values as an input and outputs a non-linear deflection corresponding to the physical properties of the branch. A least squares fit of the Euler-Bernoulli beam model is used with the physical properties encoded into c_2 and c_4 (These coefficients are discussed in detail in Section 5.2).

Since all information on how to animate a vertex must be accessible within the vertex shader, it is necessary to not only store the weight of the vertex of the current branch, but a whole hierarchy of up to 4 weights is kept in a four-component vector attribute in the vertex stream. Additionally, an index value to the current branch is stored as a float vector attribute. This allows looking up branch information in the branch data texture for the given index. As a next step the hierarchy of parent branches is reconstructed from the

branch data texture. This enables the evaluation of the whole hierarchy on a per-vertex level. The weights for all unused hierarchy levels are simply set to 0, thereby eliminating their contribution to the final vertex displacement calculation.

5.3.6 Propagation of Animation Data through Hierarchy

Weights have to be transferred from vertices on any branch down to the branches of the lower hierarchy levels. This is necessary to keep the hierarchical nature of the animation, displacing sub-branches together with their parent. Propagating weights is the last step of animation data generation for branches.

The hierarchical relationship for all branches is known, so to obtain weighting information, the vertex on the parent branch nearest to the local base point has to be found. Its weights are transferred to all vertices of the local branch and all sub-branches. With trees generated by natFX [6] or XFrog [28], child branches always emanate directly from segment edges along the cylinder, so the values of the nearest vertex can be taken directly. Of course, a bilinear weighting of the nearest parent vertices has to be used if child branches could be spawned at any position alongside a cylinder segment. In any case, the inherited weights have to correspond with the local weighting on the parent branch. Figure 5.7 illustrates how weights are propagated from the trunk down two levels of hierarchy, always keeping the weight values of the parent levels and adding a new weight for the next hierarchy level.

5.3.7 Propagation of Branch Animation Data to Leaves

Finally, all animation information calculated for the branches of a tree is propagated to the leaves to make them move together with the branches they are attached to. A very simple approach is taken by finding the branch vertex nearest to the leaf's petiole position and inheriting its weights and branch index onto all 4 vertices of the leaf. A unique animation model for leaves is layered on top of branch movement at runtime instead of adding a fifth level of hierarchy. This allows leaves not only to bend but also being twisted along the petiole axis, as is described in Chapter 5.5.2. All necessary animation data needed at runtime to perform this additional animation is taken directly from the vertex position, texture coordinate, normal and tangent, so no additional information has to be stored in the vertex stream. This makes preprocessing for leaves very fast and straight-forward.

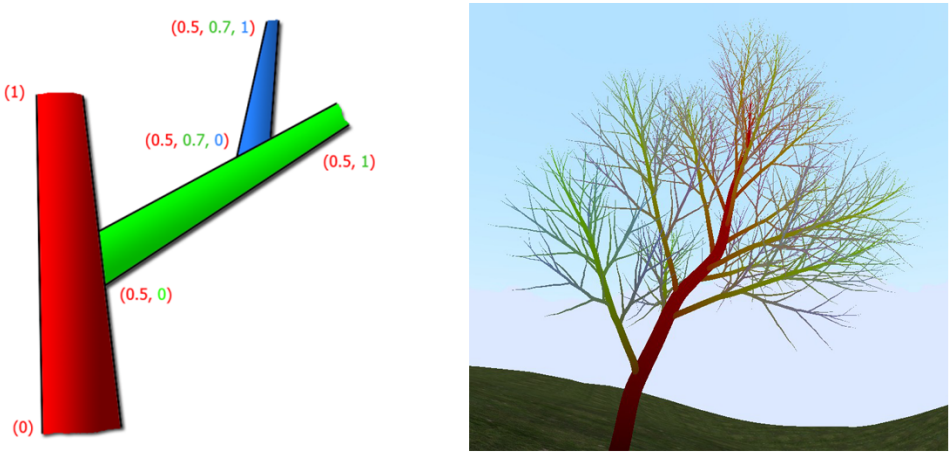

Fig. 5.7: Left: Weights are propagated from the trunk to a main branch and a sub-level branch. Right: Weights on a tree represented as color values. Red indicates the top hierarchy level, followed by green for main branches and blue for their sub-branches.

5.3.8 Simplifications

All branches of one hierarchy level are assumed to be identical in respect to their physical properties. This is a valid simplification for trees generated by tree generation packages like natFX [6] or XFrog [28], because they usually distribute child branches and leaves uniformly on a tree. This simplification is necessary because only a limited number of noise textures can be used if the system is to perform efficiently under real-time constraints. Oscillation characteristics are baked into the noise texture upon calculation from an average over all branches in a hierarchy level. The deflection amplitude is modulated per branch in respect to its total length compared to the average length of all branches on the hierarchy level. Randomly distributed offset values ensure that different points in the noise texture are sampled for each branch. Noise texture generation is discussed in detail in Section 5.4.

Weights could be adjusted individually depending on the actual branch length and number of sub-branches. Frequency and amplitude modulators could be calculated and stored in the branch data texture. However, it was found that this level of precision is not necessary, especially since a *stochastic* approach to animation is taken instead of a full simulation anyway, so a convincing *appearance* of the animation is sufficient [52, 69].

5.3.9 Animation Data Granularity

In total there are 3 levels of granularity in the preprocessed animation data:

- *Per tree*: Animation frequencies and amplitudes are constant for all branches in a tree. These values are set in a constant buffer in the vertex shader and can therefore be easily tweaked at runtime by an artist. Oscillation data is stored in a set of 4 noise textures, one for each hierarchy level.

- *Per branch*: The local branch space consisting of the principal axis $\vec{t}$ and the deflection vectors $\vec{r}$ and $\vec{s}$ has to be stored for each branch. Branch space basis vectors are stored in object coordinates. The index of the parent branch is stored, or 0 if there is no parent. Also a random offset value is stored per branch to sample the noise textures from a unique position and avoid uniform motion for all branches of a hierarchy level.

  This data is stored in a texture called the "branch data texture" with each branch placed in a column and its attributes stored in the rows. It is assumed that the number of branches does not exceed the maximum number of texels allowed in a row by the GPU, which is 8192 at the time of writing. Otherwise several rows of branches have to be stored below each other, or a texture array has to be used. Part of a branch data texture can be seen in Figure 5.8.

- *Per vertex*: An index value is stored on a per vertex level to identify which branch the vertex belongs to, together with a hierarchy of 4 weights that scale the amplitude of branch deflection for each hierarchy level. These values are written directly to the vertex stream. A total of 5 vertex attributes are therefore needed to hold the information.

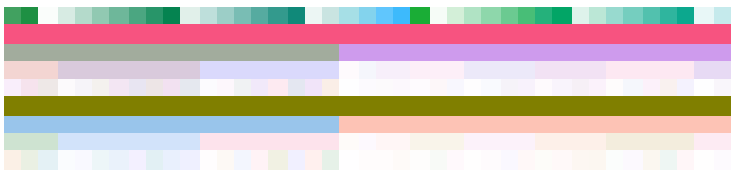

Fig. 5.8: Part of a branch data texture. Each column represents one branch. The data layout is from top to bottom: branch indices (stored for the 4 hierarchy levels in $RGBA$), noise texture offsets (stored for the 4 hierarchy levels in $RGBA$), $\vec{r}$ vectors for 4 hierarchy levels, $\vec{s}$ vectors for 4 hierarchy levels.

The three levels of granularity are also illustrated in Table 5.1. To animate the tree at runtime a global time value, noise textures to represent wind and constant values to scale motion frequencies and amplitudes have to be fed to the vertex shader. Details on generation of noise textures to stochastically reproduce damped oscillation of branches can be found in the next section. Finally, Section 5.5 describes how this animation data is used to animate the tree on a frame-to-frame basis. Because the whole branch hierarchy is available per vertex in the vertex shader and the transformation stack can be built dynamically, no joint transformation matrices or similar calculations have to be performed per branch separately. Therefore this method scales linearly with the number of vertices and independent of the number of branches.

Per tree	amplitudes, frequencies, motion textures
Per branch	$c_2, c_4, L, \vec{r}, \vec{s}, \vec{t}$, noise texture lookup, parent branch index
Per vertex	branch index, weights

Tab. 5.1: The three levels of animation data granularity.

5.4 Wind and Animation Model

Based on the work of Yung et al. [11], each branch is treated as a harmonic oscillator. Yung suggests using 2D motion textures containing displacement information to control the animation of branches swaying in wind. Constructing these textures in the frequency domain from a power spectrum containing the wind force as well as the physical properties of a branch, and then transforming the signal into time domain allows replicating the natural look of branch motion, which shows characteristic oscillation frequencies but also an aperiodic irregular movement. This complex behavior cannot be captured fully by simply superimposing sinus waves or smoothed triangle waves atop each other, as Sousa [67] does. Yung generates one noise function per tree, but for this thesis one noise texture is created per hierarchy level. Only turbulent wind motion can be represented in this way, strong directional wind is added separately as an additional deflection at runtime. Separating the two forms of wind is similar to the approaches of Ota et al. [52] and Sousa [67], representing wind as a combination of a noise function and a global directional force.

5.4.1 Uncoupled Harmonic Oscillation

The algorithm presented in this thesis treats each branch as an *uncoupled* harmonic oscillator, as Yung [11] does. A full simulation would have to take the dependence of branches on their parents and children into account. The relationships are of a very complex nature due to their dependence on the number and distributions of sub-branches and leaves. Not only does the number increase exponentially with every hierarchy level, but also the total number of children varies for each branch, thereby prohibiting optimal parallelization on the GPU. Also, wind becomes predominantly turbulent when passing through a tree, hiding the principal wind direction for slight or medium wind. All these complex effects are very expensive to evaluate. Instead of fully evaluating this system, a branch is simply considered to move with its parent's frame, being dragged along by the parent motion. Due to the predominantly turbulent nature of wind and the high complexity of the model as well as wind being invisible to the naked eye it is hard for an observer to actually see the difference between an uncoupled system of each branch moving independently, but in a coherent manner, and a full simulation [52].

This algorithm represents wind as a *power spectrum*, because for slight or medium wind no predominant wind direction can be observed directly. Even for strong wind only the turbulent part is considered for precalculation of motion data, while a directional force is superimposed as a separate step at runtime. The model to calculate the power spectrum of wind, which was also used by Yung, has been taken from Shirley and Chiu [63], with v_m being the mean wind velocity:

$$P_w(f) \propto \frac{v_m}{(1 + f/v_m)^{\frac{5}{3}}} \tag{5.4}$$

5.4.2 Generating Noise Data

A stochastic approach is taken to generate noise data for branch motion synthesis. The noise function should comply with the empirically evaluated behavior of branch motion. This means that the resonance frequency of a branch has to be found, which depends on its length and thickness and the number of sub-branches and leaves attached to it as well as the elasticity of wood.

The noise data is generated in frequency domain. A random Gaussian field is created with the wind and the oscillator response functions defining the power spectrum. Applying the inverse Fourier Transform yields the signal in time domain. A physically defined power spectrum is preferred over other noise generation techniques like Perlin noise [54] because the resonance

frequencies and a wind power spectrum can be taken into account. This leads to motion much closer to a full simulation than a purely heuristic approach.

Empirical data collected by Coder [12] is used to obtain the resonance frequency of a branch. The frequency is calculated as $2.55L^{-0.59}$ with L being the length of the branch. This value is only valid for broad leaf trees. Leafless trees or conifers have approximately 2.5 times this frequency because there is less surface area reacting to incoming wind.

5.4.3 2D Motion Textures

If a one-dimensional noise signal is created, then an individual signal is needed for each branch because the noise function should remain aperiodic so that motion of branches remains uncoupled and the resulting animation is convincing. A 2D noise texture is created to solve these problems. A 2D random Gaussian field and a 2D velocity spectrum are calculated. The power spectrum being radially symmetric allows any trajectories leading through the noise texture after transforming it into time domain to follow the defined power spectrum. As long as a trajectory does not close onto itself, the sampled signal remains aperiodic. This can be ensured by rejecting trajectories if the ratio of the x and y coordinates of the trajectory vector is a rational value. The resulting noise texture tiles seamlessly, so filtering outside the area of $[0, 1]$ still results in a coherent signal. Figure 5.9 shows an examplary trajectory through a 2D noise texture.

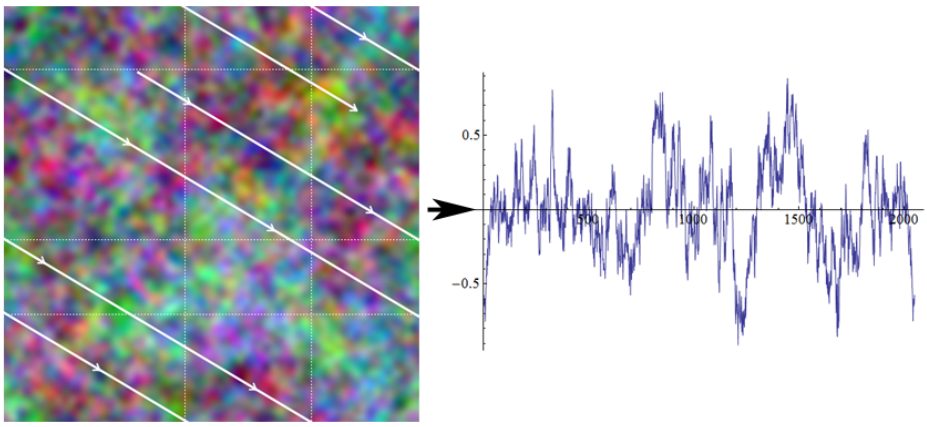

Fig. 5.9: A trajectory through the noise texture, transferred into time domain. A trajectory ensures that the sampled signal remains aperiodic. Image courtesy of Ralf Habel [30].

Fortunately, all branches of one level of hierarchy can be assumed to have roughly the same properties regarding their oscillation behavior. So instead of generating a unique noise function for each branch, only one noise field is generated for all branches of a hierarchy level. By setting an individual trajectory for each branch it is guaranteed that a unique aperiodic motion behavior can be extracted. The speed of movement along the trajectory is varied according to the length of each branch, thus effectively modulating the motion frequency.

Boundary conditions for the maximum and minimum frequency represented in the texture have to be set to retain smooth signals when crossing over texture edges. Initially, when transforming into time domain, the highest frequency captured by the texture oscillates at a length of 2 texels. To closely recreate this signal advanced sampling like *sinc* would have to be applied, but graphics hardware only allows bilinear filtering. The frequencies are reduced by padding the signal with zeros, thus effectively *smoothing* it over a larger area of texels in time domain to avoid sampling artifacts. A minimum cycle length of 8 texels is set for the highest frequency to allow a reasonable reconstruction with bilinear filtering. The lowest frequency allowed has a cycle length of one fourth of the texture resolution. Details on the generation of the noise textures can be found in [30].

5.4.4 Damping

Damping the oscillation is very important for believable branch motion. A simple per-vertex approach like the one presented by Sousa [67] accounts for damping by overlaying a set of smoothed-out triangular wave functions, thus laying damping directly into the hands of an artist. However, in the context of noise textures, damping is directly integrated into the generation of the signal as a parameter together with mass and a natural oscillation frequency [11].

Finding a correct damping value is crucial, because *underdamped* motion results in the resonance frequency being dominant and overshadowing other frequencies, while an *overdamped* oscillation hides the resonance frequency. Figure 5.10 illustrates noise textures generated for the damped and overdamped case. $1/f^{\beta}$ noise has a spectrum very close to the overdamped case and is often used to model natural phenomena [52]. However, simple $1/f^{\beta}$ noise does not account for damping effects coming from the physical properties of branches moving in wind: Branches must be able to hold up their own weight, but also have the elasticity to withstand strong winds without breaking. Moore and Maguire [48] suggest using a value in the slightly underdamped region, with findings based on empirical measurements. Larger

branches however should be placed near the overdamped case to replicate the behavior of withstanding stronger winds instead of oscillating to their own frequency.

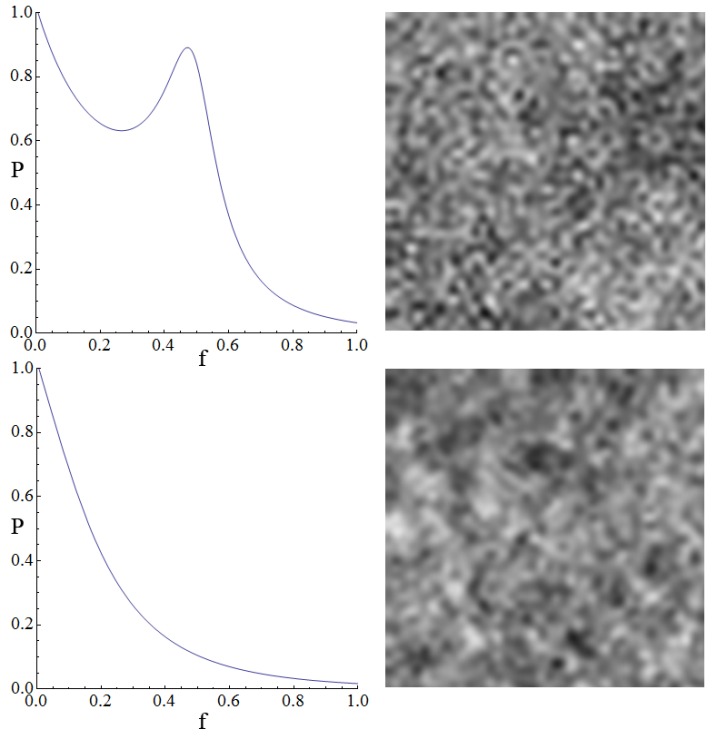

Fig. 5.10: The frequency distribution in noise textures for the damped (top) and overdamped (bottom) case. The resonance frequency is a clearly visible bump in the histogram of the damped case.

5.4.5 Limitations

One obvious drawback of a stochastical method in contrast to a full simulation is the possibility of branches to intersect each other because they are moving independently and no inter-connection is evaluated. However, when looking at a tree influenced by wind, the larger branches which shape the overall form of motion are usually at a safe distance from each other. On the other hand, the oscillation frequency of small twigs, which are mostly prone to intersection, is relativly high when compared to larger branches, so even if an intersection should happen it would occur only for a few frames and be hardly noticeable. Besides, usually a large number of leaves covers the treetop, thus blocking the view to possible intersections.

Another drawback of a stochastic method is the inability to react to outer environmental influences, for example surrounding objects limiting motion in that direction. The only solution to solve exterior influences would be to decouple any affected branches from the stochastic evaluation, either by storing them as a separate object altogether or by adding extra motion information on a per-branch level. This could be achieved by writing data to the branch data texture which contains all information needed to transform a branch at runtime. Information in this texture is also looked up when evaluating the hierarchy, so theoretically rotation or bending information could be added to the texture data. However, of course this additional data still needs to be calculated from an exterior simulation on the CPU and is therefore not fully incorporated into the stochastic method.

5.5 Animating on a Frame-By-Frame Basis

Most of the data necessary for branch deflection is precalculated, but there has to be input varying continuously each frame to create motion. Wind and the oscillating motion of branches are represented as a two-dimensional oscillation field and stored in noise textures, so animation can be applied straight-forward by displacing sample positions in these textures each frame. The data in the textures is guaranteed to correspond to the oscillating frequencies of branch movement and repeat seamlessly along all possible trajectories through texture space. Also the textures tile seamlessly when wrapping lookups around the border, so animation values exiting the texture space of $[0, 1]$ stay valid.

Having deflection vectors readily at hand from the noise textures makes calculation of the current animation step very fast and efficient. The only input needed is a global time value, treated as an *offset* to displace lookup in the noise texture. Increasing the offset continuously over time automatically results in coherent animation. Modulating the *speed* of increase corresponds linearly to an alteration of the animation frequency. Amplitudes can be modulated independently from frequencies by scaling the deflection values sampled from the noise texture.

5.5.1 Animating Branches

The branches of a tree should move in a correlated fashion while retaining their hierarchical structure. Usually, an animation system similar to vertex skinning [78, 1, 52, 60] is employed, which assigns a number of weights and branch indices (usually called “joints” in the case of vertex skinning) to each

vertex. Joints are hierarchical structural elements which are usually transformed by rotation. Due to their hierarchical nature each joint transforms all its children with it. Vertex skinning normally has up to 4 joints influencing the position of a vertex, with their weights summing up to 1. A weighted average of these matrices is calculated which is then used to finally transform the vertex position and normal. For this algorithm a *hierarchy* is encoded into the 4 weights and one branch index instead of a list of independent weights and joints, so the results are not averaged but displacements are superimposed atop each other to obtain a hierarchical transformation.

When animating the branch vertex, all four hierarchy levels are evaluated one after another directly in the vertex shader using the respective branch indices and weights. If the vertex is located on an upper level in the hierarchy, then the weights of the sub-levels are simply set to 0 to eliminate their influence on the final displacement alltogether. Dynamic branching is used in the vertex shader to speed up calculations for vertices in higher-up hierarchy levels.

Obtaining animation information for a frame

The input driving the animation each frame is reduced to two amplitude modulators, scaling the deflection along the $\vec{r}$ and $\vec{s}$ vectors of a branch. These values are sampled from the R and G channels of the noise texture. One noise texture is available per level of hierarchy, because changes in branch length and thickness result in different resonance frequencies and oscillation behavior. These deflection values can be scaled at runtime by a modulation value defined by an artist to tweak animation appearance. Figure 5.11 illustrates how branch deflections along $\vec{r}$ and $\vec{s}$ are combined to obtain the final deflection.

The deflection is performed for each level in the branch hierarchy successively in order to create a hierarchical motion. The sample point in the noise texture is determined by a fixed starting point and a time offset which changes every frame, creating a trajectory through the noise texture and causing continuous oscillating motion. The starting point is chosen randomly per branch when animation data is generated and stored in the branch data texture. The time-dependent offset value is controlled by an artist to make branch motion consistent with the desired oscillation behavior. Usually, smaller twigs oscillate faster than larger branches. The tree generation packages used to create the meshes for the demo application of this thesis show a reduction of thickness and length between two levels of hierarchy of factor 2. Therefore the speed of traversal through the noise texture was doubled for each additional level of hierarchy. The amplitude on the other hand decreases with each

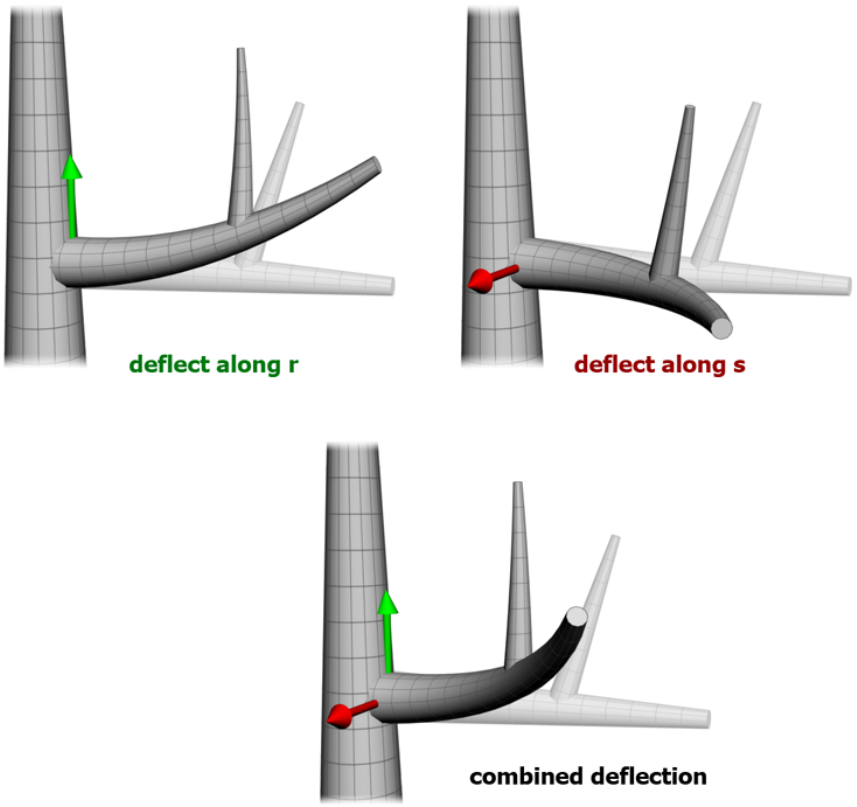

Fig. 5.11: The deflections along the $\vec{r}$ vector (top left) and the $\vec{s}$ vector (top right) are combined into the final deflection (bottom).

level, as branches become shorter and large absolute amplitude values would lead to over-bending and distortion. All of these values are transferred to the GPU as constants in the vertex shader and can be easily changed at runtime.

Modeling Wind Direction and Wind Strength

Sampling the noise textures to deflect branches only accounts for turbulent wind. Small twigs start to move even at slight wind, where turbulence is the predominant factor. Deflection mostly seems erratic and no clear wind direction can be observed from it. Only strong wind is able to bend even thicker branches into the direction of the wind, thus making the direction visible indirectly from bending. This effect has to be encoded into the animation system as well.

Wind direction is modeled as a 3D vector in world space. Branch space is defined by a principal axis along the branch, and 2 orthogonal vectors $\vec{r}$

and $\vec{s}$ in the tree's object space. Transferring the wind direction into object space allows calculating the influence of directional wind on each of these two axes by calculating the dot product of the wind direction vector and the corresponding branch vector. The length of the wind direction vector corresponds to the desired impact of directional wind onto the final bending appearance and usually depends linearly on wind strength. $\vec{r}$ and $\vec{s}$ are assumed to be normalized. Thus, evaluation of the dot product results in branches orthogonal to the wind direction being influenced significantly more than branches headed parallel to the wind direction, with the maximum being the wind direction vector's length. Negative results of the dot product are also valid because they are needed to bend a branch into the opposite direction.

The result of the dot products is added onto the two swaying amplitudes along $\vec{r}$ and $\vec{s}$ originally emanating from noise texture lookups, thus increasing the deflection into the direction of the wind. The swaying frequency remains unaffected. The importance of re-scaling the length of a branch becomes clearly visible especially with stronger wind when branches would be elongated if simply sheared into the wind direction. Also, since turbulent wind still is present in the calculations, branches do not point into wind direction permanently but bend into other directions as well. However, the main wind direction becomes clearly visible for strong wind.

Wind strength for turbulent wind is modeled as a simple scalar value which directly modulates the amplitude of deflection. The influence of wind strength increases with each level of hierarchy so smaller twigs are influenced by even slight wind. Since thickness and length of each hierarchy level are found to be roughly half of the next upper level the influence is increasing by a factor of 2.

5.5.2 Animating Individual Leaves

Each leaf should react individually to wind in addition to moving with the branch it is attached to. As can be seen in nature, already small gusts of wind make leaves move erratically in different directions uncoupled from wind direction similar to the behavior of small twigs. Leaves are much lighter and have a larger surface area to capture incoming wind, so their oscillation frequencies are even higher. Also, leaves bend in different angles and are twisted around the petiole axis, resulting in a *torsion*. (This behavior of course only applies to broad leaf trees.) Leaves which are close together spatially are expected to move at a similar frequency and intensity, although not necessarily in the same direction. As a gust of wind moves through the tree it is expected to influence the leaves one after the other, so the wind

direction can be tracked indirectly by watching waves of leaf motion moving through the treetop.

A unique model for leaf animation is used to capture this behavior instead of just adding an additional branch hierarchy level. The leaf's tangent space, texture coordinate values and vertex positions are used directly for a simple animation approach. Using the vertex position as an input value complies ideally with the effect of nearby leaves moving in a correlated fashion.

Motion is divided into lateral motion with vertices being displaced along the tangent or normal vectors, and a rotational torsion with vertices rotated around the petiole. Calculation of the rotation components is straightforward if the petiole is assumed to be at the center of the leaf texture. Figure 5.12 illustrates torsion along the normal and tangent vectors.

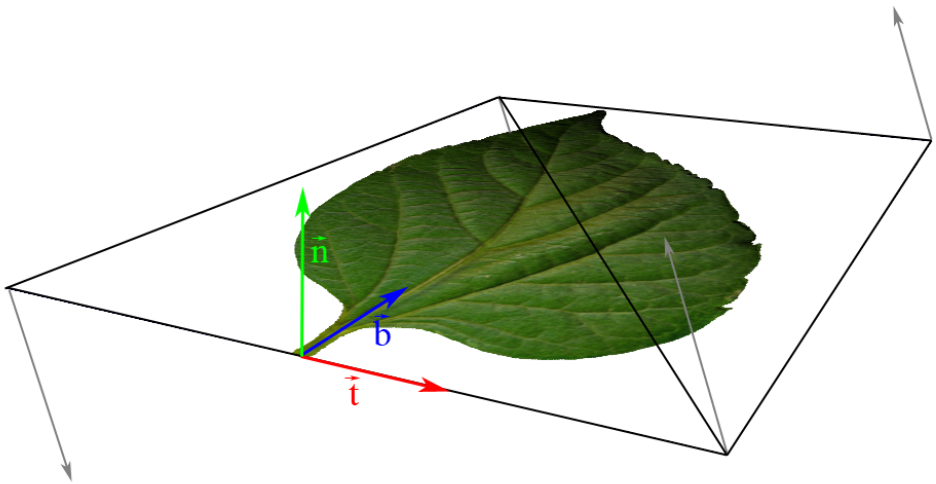

Fig. 5.12: The end points of leaves are displaced along the normal and tangent, resulting in a torsion. Image courtesy of Ralf Habel [30].

Obtaining Animation Information for Each Frame

An additional cube noise texture is used for leaf animation, and the x, y and z coordinates of the leaf vertex position are used as weighted sampling coordinates in the texture. The lookup is displaced by the wind direction vector scaled over time, effectively moving the noise texture through the treetop over time continuously. Similar to handling branch motion, the texture is filtered to avoid too high frequencies, which would cause the individual vertices of one leaf to behave too differently. In the noise texture the maximum frequency takes 8 pixels for one oscillation. The three individual degrees of

motion are stored as RGB values in the noise texture. The R and G channels hold information for lateral movement along the tangent and normal plane of the leaf, and torsion is stored in the B channel. Although this approach is quite simple, it produces satisfactory results. One reason is that leaves are already moving with the branches, so the noise texture lookups are already pre-displaced by a modified vertex starting position. This effectively avoids a homogeneous motion rolling through the tree top, which could be witnessed if the branches stayed absolutely still. Additionally, since slightly different noise texture samples are taken for each vertex of a leaf, the leaves automatically bend and twist in a believable manner and do not remain rigid.

Of course this approach could also be expanded to leaves of higher geometrical detail as the motion texture still is filtered uniformly over the area of a leaf. The animation of a leaf remains consistent as long as its size stays inside the texture space area associated with the highest frequency.

5.5.3 Intuitive Parameter Set

Amplitude and frequency for branches and leaves are the values which shape the appearance of the animation. The noise textures on the other hand could be re-used for similar types of trees which have roughly the same oscillation characteristics. Motion amplitudes and frequencies are found by experimentation, starting with values from Coder [12]. Because they are only used as modulation input in the vertex shader, these values can be changed at runtime each frame which makes tweaking fast and easy.

- A change in *amplitude* is simply a multiplication factor for the amount of deflection, thus directly influencing the amount of vertex displacement. This corresponds to u in the Euler-Bernoulli beam model seen in Section 5.2.

- A change in *frequency* modulates the speed at which the offset value is increased each frame for noise texture lookups. This correlates linearly to the speed of branch swaying.

Besides an amplitude and frequency value associated with branches and leaves there is of course *wind direction* and *wind strength*. Wind strength again modulates the trajectory speed in the noise texture, and wind direction is used for the directional wind component of the algorithm only. These are all the values needed to modulate the appearance of the animation. The parameter set stays small and intuitive, as changes can be made and observed at runtime. Also changes to the individual levels of hierarchy stay *local* to that level, so animation properties for each level can be changed without

influencing the overall appearance of the animation. Figure 5.13 illustrates a GUI to modify all these parameters in real-time.

Fig. 5.13: A GUI to change all parameters influencing the animation behavior at runtime.

Of course, an additional variation has to be accounted for if a larger number of trees is to be rendered at once. An offset value for noise texture lookup can be added per tree to increase the variety of the perceived animation. Also, an additional "global" wind noise texture could be used and moved along the XZ plane in wind direction to modulate wind direction and strength for each tree over time in a coherent fashion.

5.5.4 Scalable Complexity

The complexity of the animation system can be scaled according to the desired precision to save rendering time. This is especially important if this technique is to be combined with an LOD approach where far-away trees do not need full shading or animation detail.

The model proposed in this thesis displaces not only the vertex *position* like Sousa [67] does, but also rotates the vertex normal and tangent to retain

an orthogonal tangent space necessary for advanced per-pixel lighting calculations. If a tree is farther away from the viewpoint and does not need a complex per-pixel illumination model, then it is also feasible to ignore normal and tangent transformations. Besides, motion amplitudes are small values and vertex displacement stays on a small scale with slight or medium wind. Normal rotation therefore would also remain at a few degrees, thus making missing tangent space corrections possibly not visible at all.

Another possible simplification is the reduction of the number of hierarchy levels. If no small twigs on the fourth hierarchy level are rendered at all, for example because they are already baked into billboard clouds, then there is no need to store branch indices and weight values for these hierarchy levels. Reducing the amount of data stored for a vertex in the vertex stream additionally makes caching on the GPU more effective. About 30 ALU instructions in the vertex shader are needed per hierarchy level to calculate transformations including normal and tangent rotation, so simply reducing the number of evaluated levels of hierarchy already reduces work in the vertex shader.

Chapter 6

Implementation and Results

6.1 Overview

A DirectX 10 proof-of-concept implementation was created to show that trees of a high level of detail can be rendered and animated in real time by the methods proposed in this thesis. The implementation uses state-of-the-art rendering techniques. The whole scene is rendered in HDR (High Dynamic Range) and then tonemapped to the RGB color range. Bloom is used to create the effect of overbright spots in the image bleeding out to other regions, and light shafts are included to enhance the visual depth of the image. The modular approach of a strict separation of offline preprocessing wih specialized tools and a runtime system only based on vertex and pixel shaders makes adaptation of the techniques to any modern shader-based rendering pipeline easy and straight-forward.

Section 6.2 deals with obtaining tree data and how it was preprocessed to be loaded by the demo application.

The pipeline of the demo application is described in detail in Section 6.3. The individual steps taken to render a frame and how they are combined is discussed in detail in this section.

Finally, rendering statistics and performance characteristics are presented in Section 6.4. These numbers indicate that all of the advanced techniques presented in this thesis can be integrated into a full rendering pipeline to run at 60 frames per second. The techniques were also tested on a larger number of trees, up to 100 visible at once.

6.2 Tree Import and Data Processing

As a first step, a tree model has to be generated. The tree generation package XFrog [28] was employed to create a tree mesh in Autodesk Maya [3]. Only vertex positions, normals and texture coordinates are exported from Maya. A separate tool is used to calculate the tangent space and process the per-

vertex ambient occlusion data. Animation data is created offline by the demo application if necessary.

All information needed at runtime is loaded from a "tree definition file", which is basically an *INI* file holding all data for rendering and animation parameters. This includes links to tree geometry, animation data and textures for leaves and branches. It is easy to integrate new trees by writing new tree definition files and then loading them into the demo application at runtime.

6.2.1 Tree Generation in Autodesk Maya

XFrog [28] was used to generate trees in Autodesk Maya [3]. The selected level of detail includes 4 levels of hierarchy for the branching structure and leaves represented as textured quads. Two different tree models were generated to test different performance characteristics for smaller trees with very dense treetops and larger trees with fewer but larger leaves and a sparse treetop. The generated models have approximately 70,000 triangles and 100,000 vertices each. Both trees feature over 10,000 leaves. Leaves are individual textured quads, but obviously a tree generation package like *XFrog* would be suited to generate a series of LODs (levels of detail) and integrate them into the data generation pipeline. However, LOD techniques are beyond the scope of this thesis and most of the time only one tree is rendered at close distance to observe rendering and animation details.

The models were exported in *OBJ* file format, so geometry is stored as lists of triangles. Branch geometry and leaves are stored as two separate groups in the *OBJ* file. The human-readable *OBJ* format was chosen because parsing is fast and straight-forward. Only the most basic information of vertex positions, normals and texture coordinates are stored. This is all data necessary, because tangent space can be easily calculated separately and the per-vertex ambient occlusion has to be created with a specialized tool anyway. Animation data also has to be calculated separately.

6.2.2 Tangent Space Generation

Tangent space is calculated in a spezialized tool. The positions and texture coordinates coming from the *OBJ* file are used as input with the assumption that textures are always mapped straight onto the geometry according to a local right-handed coordinate system and not flipped along the u or v axis. A Gram-Schmidt orthogonalization is performed to guarantee an orthogonal tangent space even if the original texture coordinate distribution was not uniform. An algorithm by Lengyel [38] was used to generate the tangent

space. The data is finally written to another *OBJ* file containing one tangent vector per vertex.

6.2.3 Ambient Occlusion Processing

Ambient occlusion calculation is performed by the *GUPRT* tool set which was created by Ralf Habel to evaluate precomputed radiance transfer for other research purposes [31]. The tool however also allows simple ambient occlusion calculation and is able to take light transmittance coming from the leaf translucency and HL2 maps into account. Evaluation is done on a per-vertex level because the tree mesh is detailed enough to assume a sample point for a range of at maximum 20 centimeters, which is sufficient precision for ambient occlusion. Color values are linearly interpolated over the triangle area upon rendering.

The *GUPRT* tool outputs the newly generated ambient occlusion colors as an additional *OBJ* file.

6.2.4 Geometry File Format

After the previous steps, vertex geometry to represent one tree mesh is distributed among 3 *OBJ* files. One file stores vertex positions, normals and texture coordinates, exported from Autodesk Maya [3]. Another file contains tangent vectors, and the third one holds the color information for ambient occlusion.

Geometry is now converted into a proprietary binary file format. Most common existing geometry formats like *FBX* [2] store a set of individual vertex indices for position data, normals data and texture coordinates of each primitive vertex. While this reduces file size somewhat and may be useful for 3D content creation tools like Autodesk Maya [3], it is not well suited for real-time rendering. Graphics hardware demands only one index per primitive vertex to render, and vertex data has to be located in one large continuous buffer with data for all vertices lined up. So when using a file format like *FBX*, mesh data has to be processed after loading to convert a number of individual indices and individual data streams to one index per primitive vertex and one continuous data stream.

The binary file format called *VBO* (for "Vertex Buffer Objects", a term from OpenGL equivalent to "Vertex Streams" in Direct3D) stores geometry data directly in the required format. Vertex data can be loaded from the file as one buffer and uploaded to the GPU directly without any additional preprocessing. Data can be stored either as a list of individual streams for

each vertex attribute or as one interleaved vertex stream containing all vertex attributes and ordered by vertices. However, no noticeable performance changes in the demo application could be found when switching from one representation to the other.

The file format supports an arbitrary number of attributes per vertex, with each attribute ranging from a one-dimensional float value to a four-dimensional float vector. Each attribute is identified by a string name which is used to bind the stream to an attribute in the vertex shader. This is done when generating the vertex stream and uploading it to the GPU at load time. Semantics, which are an integral part of the Direct3D *FX* format, are used to match vertex attributes to shader input streams.

Additionally, one index stream for the whole VBO and the primitive type to be rendered are stored in the file. Usually, primitives are a list of triangles but triangle strips are supported as well.

6.2.5 Tree Definition Files

A tree is loaded into the demo application via a tree definition file. These are human-readable *INI* files which contain all information necessary to render and animate a tree. The data stored in a tree definition file is the following:

- *Geometry*: Information to the location of the *VBO* files for branch and leaf geometry is stored in the tree definition. File locations are specified for geometry with and without per-vertex animation data baked into the vertex stream (see Section 5.3 for which data is stored per vertex). If the vertex streams containing animation data are not found on the hard disk then they are automatically calculated from the static geometry when the tree definition file is loaded. The newly created vertex streams as well as the branch data texture are saved to the file locations specified in the tree definition file.

- *Animation data*: The location of the branch data texture and a set of noise textures are stored as well as a set of constants that define initial branch and leaf deflection amplitudes and frequencies. These constants can be changed at runtime via a GUI to tweak the appearance of the animation.

- *Textures*: A set of textures to render leaves and branches is specified. This data set consists of the 8 textures necessary to render a leaf with advanced translucency and a set of an albedo map, a normal map and a displacement map for the branches.

- *Lighting*: Lighting constants for scaling of ambient, diffuse, specular and translucent terms of branches and leaves are also specified explicitely to fine-tune the appearance of different types of trees.

6.2.6 Animation Data Generation

The noise textures are created in a custom tool which takes the average thickness and length of a branch as input and outputs a 2D noise texture. A set of noise textures is created from this tool with input values for each hierarchy level taken directly from XFrog [28]. This guarantees that the oscillation characteristics generated in the noise textures always match the underlying geometry data.

Per-vertex data is stored directly in a vertex stream. Generation of per-vertex and per-branch animation data is automatically triggered if the *VBO* file specified to hold the animation information is not found on the hard disk when the tree definition is loaded.

6.3 Rendering Pipeline

This section presents a detailed view of the demo application's rendering pipeline. The pipeline was created with state-of-the-art real-time rendering techniques to create convincing results and test the performance of the algorithms with only part of the rendering resources dedicated to them. HDR (High Dynamic Range) rendering was chosen and a custom tone mapper was applied which is based on Reinhard's tone mapper [59]. Additional settings to adjust contrast, saturation and color shift are added to enable the user to perform simple post processing steps similar to color correction in movie post-production. Shifting color and saturation allows setting a mood for the resulting images similar to the use in film and state-of-the-art games, as these are the target applications for the techniques presented in this thesis. Although color and saturation are shifted away from a natural look, results can seem even more convincing because viewers have already become accustomed to established motives from cinematography.

Figure 6.1 provides an overview of the rendering pipeline, with the main render targets used when rendering a frame displayed in different colors. The demo application renders one "scene" at a time, with all visible objects contained in this scene subsequently called "scene objects".

The rendering pipeline mainly consists of the following passes:

- *Render into Shadow Map*: The shadow map is updated each frame. It is used to test whether sunlight reaches a point on a per-pixel level.

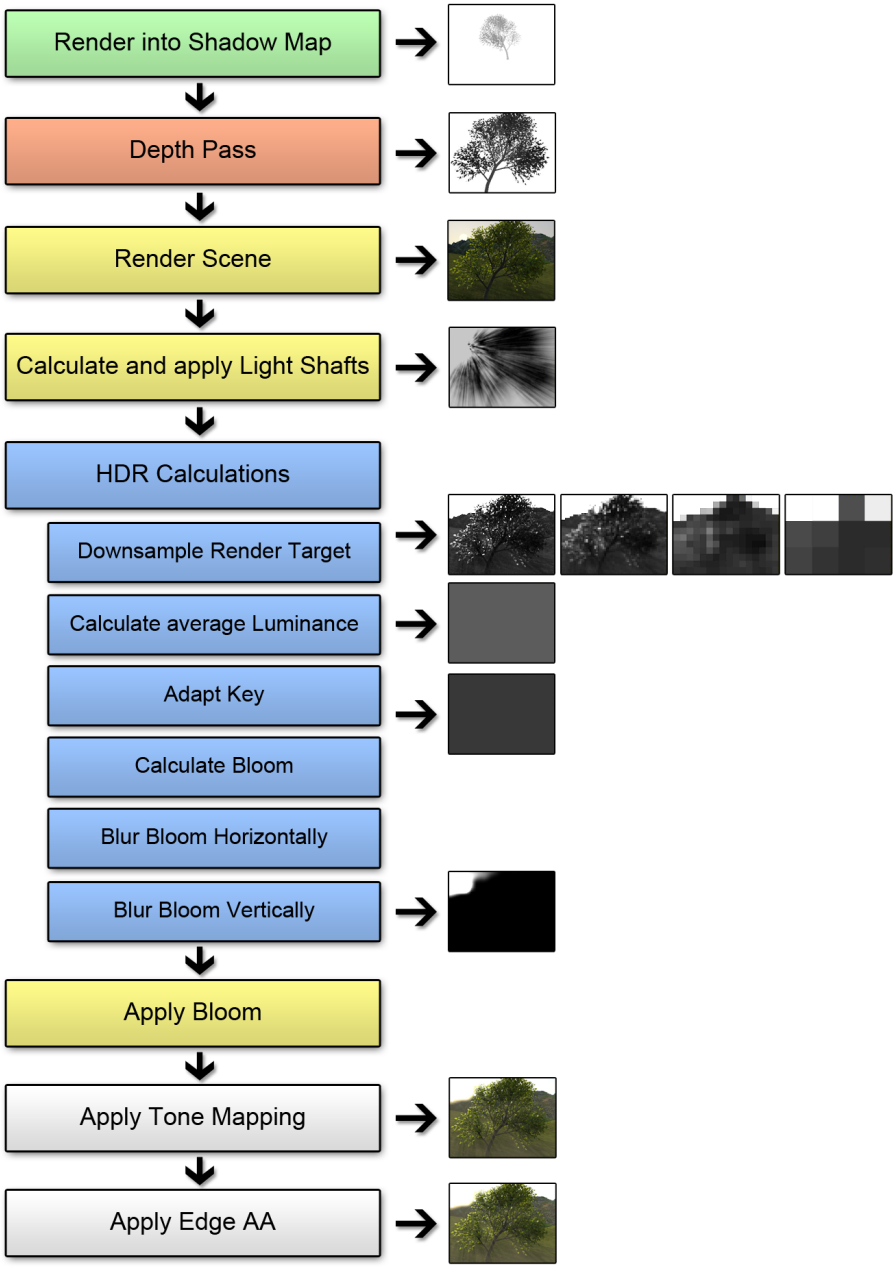

Fig. 6.1: The rendering pipeline. The most important render targets used in the respective passes are color-coded: The shadow map (green), the linear scene depth (red), the HDR scene render target (yellow), various render targets associated with HDR and tonemapping (blue) and the frame buffer (white).

Details on shadow map rendering are found in Section 6.3.1.

- *Depth-first pass*: Instead of starting with rendering all scene objects with full shaders, they are first rendered as depth values into the linear scene depth target. This allows to establish the depth buffer and use early-Z-out optimizations on the GPU when rendering the objects with their full shaders later on. This not only eliminates overdraw when rendering leaves with a complex per-pixel shader, but linear scene depth is also needed for other calculations. The depth-first pass is described in detail in Section 6.3.2.

- *Render scene*: The established depth buffer is re-used and objects are rendered with their full shaders. This includes rendering the sky, the ground, and vegetation objects. The sky is rendered with the Preetham Skylight model [56], which is discussed in Section 6.3.5. Colors are represented as HDR floating point values.

- *Calculate and apply light shafts*: Light shafts are calculated from the linear scene depth emanating from the depth-first pass. Light shafts come from a grayscale image with all pixels of the sky rendered white and being extruded away from the sun multiple times and then additively blended onto the scene. Light shafts calculation is described in Section 6.3.3.

- *HDR calculations*: The average luminance of the image and a new key value are calculated to simulate adaption of the virtual camera to changing lighting conditions. A bloom texture is calculated to create the effect of overbright areas bleeding over into the rest of the image and additively layered on top of the scene. A modified Reinhard tone mapper [59] is applied to convert the HDR values into the RGB color space. Details of this step are found in Section 6.3.4.

- *Edge AA*: Finally edge antialiasing is applied, which is a simple image-based blur approach to soften the outlines of objects. The details of edge AA are described in Section 6.3.6.

- *GUI*: A GUI can be rendered on top of everything else to adjust lighting or animations parameters. GUI visibility can be toggled by the press of a button.

6.3.1 Shadow Mapping

As a first pass the scene is rendered from the point of view of the light source. Orthographic projection is used because sunlight is modeled as a directional light. A scale transformation is applied to the projection matrix to always enclose all visible objects in the scene, in case that a larger number of trees has to be rendered. The shadow map is used later to determine the intensity of direct illumination on a per-pixel level.

Linear depth values are stored in the shadow map, with unit length corresponding to 1 meter. This allows using a simple constant offset bias when doing depth comparison to estimate if a pixel lies in shadow.

The texels of the shadow map are uniformly distributed, no advanced image warping techniques like LispSM [77] or Perspective Shadow Maps [70] were used. A uniform distribution suffices for the relatively small enclosed space which has to be covered by the shadow map. In the case of one tree the area includes only 15m on the XZ plane. The resolution of the shadow map is set to 2048 by default, although the size can be changed in an *INI* file. A resolution of 2048 was found to suffice to capture the details of each individual leaf and thin twigs. Lower texture resolutions introduced noticeable flickering effects when leaves and branches are moved by wind.

Filtering

Percentage closer filtering (PCF) [58] is used for the depth comparison between the depth stored in the shadow map and the distance of the current fragment to the light source. PCF has to be available in hardware for Direct3D 10-compatible graphics hardware if a 32bit floating point render target is chosen as input. By default 12 PCF samples are looked up for each pixel to be tested for the intensity of direct illumination, with the samples placed according to a Poisson distribution on a unit size disk surrounding the projected fragment position [58]. Because all samples stay in the area of one texel, the lookup still results in precise shadows of the accuracy of one texel. The typical artifacts coming from one PCF lookup (making individual texels discernable by gradients in u and v direction of the shadow map) are avoided by a larger number of lookups. Filtering modes can be switched at runtime, allowing either one single PCF lookup, 12 Poisson-filtered PCF lookups or Variance Shadow Mapping (VSM) [18].

The precision of the shadow map render target is 32bit per pixel. While 16bit would suffice for linear depth values, 32bit gives the advantage of PCF filtering directly by the graphics hardware without the need to implement bilinear interpolation of shadow mapping results in the pixel shader manually.

Also, 32bit precision is needed when VSM is applied, because squared depth values have to be stored in a separate channel of the render target, too. 16bit floats do not suffice to store meaningful data for squared depth values.

Performance

Interestingly, performance decreases noticeably when VSM is activated, which can be traced to the *blurring* which has to be performed in the shadow map render target. VSM works by treating the set of a linear and squared depth value of a pixel in the shadow map not as one discrete value but as a distribution of depth values. Filtering has to be performed in the shadow map onto an area surrounding the texel to achieve a valid distribution. Usually this is done by a separable blur, which blurs depth values first along one and then along the other axis. Separable blur is a valid simplification to sampling a full two-dimensional filter kernel if the kernel is radially symmetric. The performance decrease can be explained by needing 64bit per pixel in the shadow map for 2 channels of 32bit floating point precision each. Performance characteristics for different shadow mapping algorithms can be found in Section 6.4.

The size of the kernel directly influences the sharpness of the results of VSM. A 5x5 kernel was chosen for the demo application, as larger kernels tend to blur the results too much, making individual leaves disappear. This does not correspond to the behavior found in nature, where direct illumination of sunlight casts detailed shadows of each leaf onto the ground.

Poisson filtering was deemed the most suitable shadow mapping filtering technique for the rendering of precise shadows of highly-detailed geometry, both in regard to the quality of the resulting image as well as in regard to the performance impact. Because the complex per-pixel lighting shaders used in the demo application are mostly limited by performance of the ALU instructions, there proved to be no discernable framerate impact if the number of texture sampling instructions increases from 1 PCF lookup to 12 lookups. This is due to TEX instructions being performed in parallel to ALU instructions on modern graphics hardware.

6.3.2 Depth-First Pass

The depth-first pass is used for multiple purposes: First of all it establishes the depth buffer to avoid overdraw in subsequent passes. This is especially important because very complex per-pixel shaders are used to render the leaves of a tree. By having the depth buffer already available, *early-Z-out* is used by the GPU to reject most unnecessary pixels from being rendered

before the pixel shader has to be evaluated at all. Only the pixels of the nearest leaves have to be rendered fully. Of course, because alpha testing is used, leaf quads can not be fully rejected at once, so a certain amount of overdraw cannot be avoided for the pixels with alpha values below 0.5, which are then rejected *during* pixel shader evaluation instead of *beforehand*. However, using a depth-first pass still does result in a noticeably higher frame rate.

Besides rendering into the depth buffer, depth values are also stored into a 16bit floating point render target. Depth values are stored as linear values with unit length corresponding to 1 meter. The same shaders which are already used to render into the shadow map can be re-used.

The linear scene depth is needed for edge antialiasing, which is discussed in Section 6.3.6, and to calculate light shafts, which is described in the following section. Additional post-process effects like Depth of Field could also be easily achieved with a linear scene depth buffer [61].

6.3.3 Light Shafts

The effect of light shafts comes from particles in the air scattering incoming light towards the eye. Because less light is scattered in shadowy regions, an effect of shafts of light extruded from solid objects in the scene into thin air is created. This effect is often seen in film and games for dramatic purposes and increases the sensation of the image having a volume instead of being only two-dimensional [45].

The linear scene depth render target is used as an input to calculate light shafts for the demo application. Sky distance is assumed to be at 1,000 meters distance. This value is much larger than the far plane defined in the projection matrix, which is currently at 100 meters. This allows fast separation of pixels storing scene objects and pixels belonging to the sky. The basis texture for light shafts is calculated from the linear depth render target with all pixels belonging to the sky colored white and the rest colored black. A simple 8bit grayscale render target is used for this, which is also decreased in size by 1/4 when compared to the frame buffer. This makes access to the texture a very fast process. Also, the simplification does not limit the visual quality of the resulting light shafts.

In order to create the effect of three-dimensional light shafts from the black-and-white basis texture, the bright areas have to be expanded away from the sun. This is done by calculating the position of the sun in screen space and then using a vector from the sun position to the current fragment position as extrusion vector. 64 samples are taken along the extrusion vector, looking up brightness values from the light shafts basis texture along the way.

Summing up the results leads to the effect of all pixels previously white to be extruded into the black areas, as can be seen in Figure 6.2. This technique is similar to a method introduced by Kenny Mitchell in *GPU Gems 3* [46].

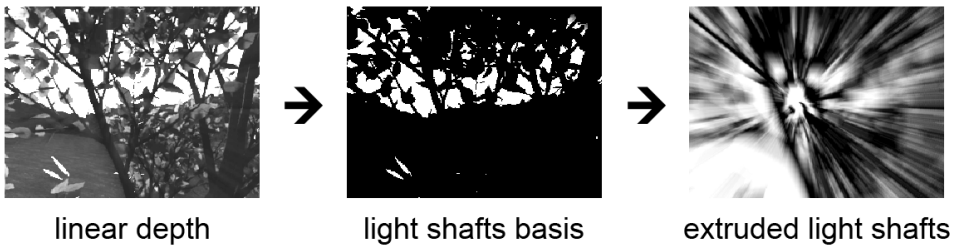

Fig. 6.2: The light shafts basis texture (center) is calculated from the linear scene depth (left). Sampling the texture multiple times on a trajectory facing away from the sun results in visible shafts (right).

Finally the resulting image is modulated by the sunlight color and additively blended onto the scene. Because this is only a post process effect instead of a full geometry extrusion, it can only be layered on top of the scene after rendering all solid objects. The light source has to be *behind* all objects in the depth buffer, which fortunately is no problem with the sun. The subtle difference of an image rendered with and without light shafts can be seen in Figure 6.3.

Fig. 6.3: A scene without (left) and with (right) light shafts.

6.3.4 HDR Rendering and Bloom

Scene illumination is rendered at high dynamic range. A Direct3D 10 32bit floating point format is used, which stores the R, G and B channel in 10

bits each and leaves 2 bits for the alpha channel, which is not used. No real physical units were used for luminance, so 10bit precision per channel suffices.

Tone Mapping

A benefit of high dynamic range rendering is the automatic adaptation to changing lighting conditions. This is mostly noticeable if the camera moves inside a treetop. Luminance values are low because almost no direct sunlight from outside reaches the inside, and also indirect illumination coming from ambient occlusion is reduced by multiple layers of leaves blocking off environment light. High dynamic range rendering is needed to still retain useful images for this low-light situation. Reinhard's tone mapping algorithm [59] was used to convert values from HDR space to meaningful RGB values.

Downsampling is performed to calculate the average luminance of the image. Each downsampling level reduces the width and height of the render target by 4. This is achieved by applying a 2x2 box filter on 2x2 samples from the image which are already bilinearly filtered from 2x2 pixels each. This leads to a 1x1 render target after 5 iteration steps if the source frame buffer has a size of 1024x768 pixels. Figure 6.4 illustrates how the color values of 4x4 pixels are averaged in one pass.

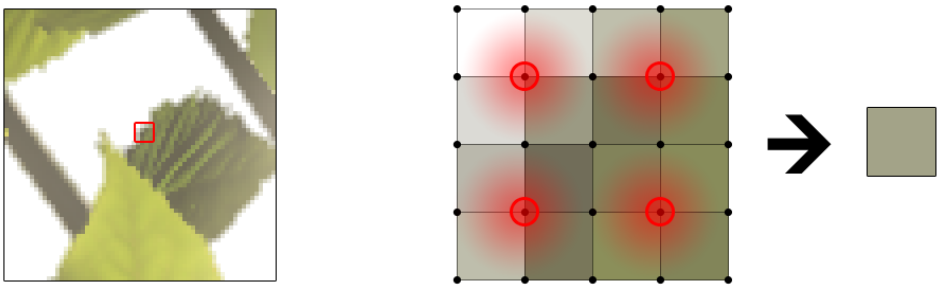

Fig. 6.4: Downsampling is performed by averaging 4 bilinearly filtered lookups.

Instead of relying on physical units, luminance values for the ground, branches, leaves and the sky were chosen carefully by hand to replicate the behavior of almost no bloom if the average luminance in the scene is high and bloom getting more appearant at low average luminance. This is especially effective if bright spots from the sky can be seen through layers of leaves while the camera is placed inside a treetop.

Bloom

Bloom is calculated as a simple separable blur from the overbright areas of the image. These regions are first rendered into a separate render target with the rest of the image remaining black. A user-defined cutoff value specifies exactly what minimum luminance is needed for a region to start blooming. However, the bloom itself is an RGB render target with color values taken from the scene. The image is first blurred horizontally and the results then blurred vertically to replicate a 9x9 Gaussian blur kernel [44]. Instead of using the full size frame buffer as bloom source texture, an already downsampled version is used to reduce the amount of pixel calculations needed and at the same time increase the size of blurring in screen space.

The blurred bloom texture is blended additively onto the scene before tonemapping is applied. The bloom kernel was intentionally kept small to avoid blooming becoming the dominant feature in the final image composition and overshadowing small details. Because the bloom still retains color values for the R, G and B channel, not only the *luminance* but also the *chromaticity* of the overbright regions bleeds over to the rest of the image. Figure 6.5 illustrates the effect of blooming, with the bloom having a subtle blueish color which corresponds to the sky color.

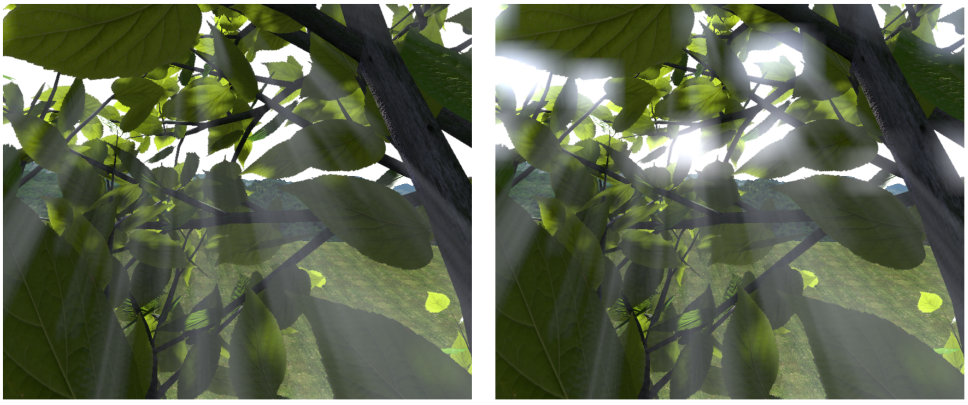

Fig. 6.5: A scene without (left) and with (right) bloom applied.

Saturation and Color Shift

Finally, a saturation modulation and a color shift are applied to the image. This was added to enable post-production effects similar to what can be seen in movies and state-of-the-art games. Key calculation can be changed to create high-key or low-key images. Figure 6.6 illustrates how different

saturation values and a slight color shift can change the impression of an otherwise identical image.

Fig. 6.6: Different saturation and color shift setups: Reduced saturation and a color shift towards blue produces a colder image (left), standard saturation (center) and an oversaturated image result in cartoonish colors(right).

6.3.5 Preetham Skylight Model

A hemispheric sky dome mesh represents the surrounding sky. Normals of each vertex in the sky dome are used to evaluate a gradient color value. Colors are then interpolated over the triangle area. A geospheric vertex distribution was chosen when creating the mesh in Autodesk Maya [3]. A geosphere is created from subsequently subdividing a tetrahedron or a similar geometric shape. This ensures an almost uniform distance between all neighboring vertices in the resulting mesh. The sky dome mesh can be seen in Figure 6.7. This guarantees a consistent image quality over the whole sky dome when rendering the mesh with interpolated vertex colors.

The Preetham Skylight model [56] is used to calculate the vertex colors. The model uses fitted simulation data from Nishita et al. [50] and Perez et al. [53] and is controlled by the sun's position, which is represented by an azimuthal angle and the angle to the zenith, the viewing direction, which is approximated by the normal of the current sky dome vertex, and an atomspheric turbidity value which represents atmasphere haziness. The turbidity is set to 3.0 and can be changed at runtime. Colors are returned as high dynamic range values with a luminance distribution corresponding to physical units. The illumination values are scaled to match the intensity values used for rendering of the rest of the scene and to have a consistent color range for the chosen HDR tone mapper to work optimally.

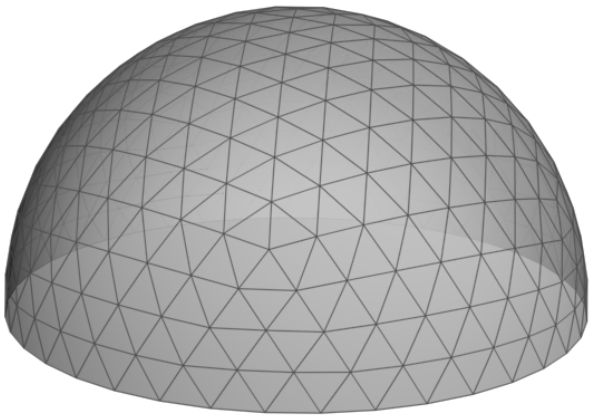

Fig. 6.7: The sky dome mesh used in the demo application. A consistent vertex distance is necessary for convincing shading.

Still sky color values are greater than ambient and diffuse values of the ground, resulting in subtle blooming of the sky. This effect is increased if the camera is moved inside a treetop where the average luminance is much lower due to a decreasing amount of direct illumination. In this case the whole sky glows bright, as can be seen in nature when looking at a clearing from within a forest.

The Preetham skylight model does not account for the sun explicitly. Therefore, the sun is rendered separately as a disk texturemapped onto a quad. The luminance is much brighter than the surrounding sky, leading to looking into the sun making the rest of the scene appearing almost black.

6.3.6 Edge Antialiasing

Fullscreen antialiasing (FSAA) is a common way to soften polygon edges in real-time computer graphics. Instead of full *supersampling*, where multiple color values are calculated for each pixel and then averaged, *multisampling* does this only for pixels at the outer edges of polygons, thus limiting the impact on performance. Both for supersampling and multisampling a number of samples must be available in the render target for each pixel to allow averaging, effectively increasing the memory requirements for color and depth buffers. If no antialiasing is used at all, then the outlines of objects are rendered as hard pixel-aligned shapes, resulting in stair-step artifacts which make the individual pixels of the frame buffer cleary visible. This can easily

destroy the believability of a computer-generated image.

However, many of the polygons in an outdoor vegetation scene are billboards which are already a geometric simplification themselves and rendered with alpha-testing, rejecting any pixels with a texture alpha value below a certain threshold. Simple multisampling cannot account for soften these edges because they are not at the outer edges of the polygon but inside the polygon area. There are techniques like "Transparency Antialiasing" which try to deal with this effect, but again need multiple samples per pixel in the color and depth buffers.

Instead of using FSAA, a technique called "edge antialiasing" (Edge AA) was employed for the demo application, which is also used by Mittring [47] in "Crysis" [26] to create convincing results while simultaneously saving graphics hardware resources. This technique works as a screen-space post-processing effect and *blurs* areas of the screen with large discrepancies in depth values within a specified kernel. This results in blurring of the outlines of objects and destroys fine detail in that area, but no multiple samples have to be stored per pixel in the color or depth render targets. A simple 2x2 box filter was used in the demo application making Edge AA very fast to evaluate and still resulting in a convincing appearance similar to multisampling, especially if the user does not look closely at the influenced pixels with a magnification tool. The results can be seen in Figure 6.8.

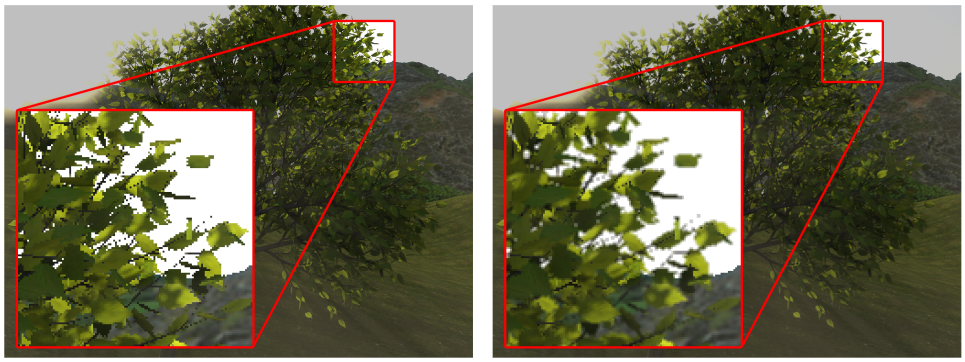

Fig. 6.8: A scene without (left) and with (right) edge antialiasing.

6.3.7 Additional Functionality of the Demo Application

A GUI can be rendered as an overlay over the scene to allow tweaking of the animation settings for wind and individual branch and leaf motion frequencies and amplitudes. Also, the light direction can be set inside the GUI. It is illustrated in Figure 5.13.

Multiple trees can be rendered at once to measure their impact on performance. Currently no LOD techniques are employed so all trees are rendered at full detail. Also, no frustum culling is enabled so all vertices of all tree meshes are transformed each frame regardless of their visibility.

It is possible to record camera paths through the scene manually or import them from Autodesk Maya [3]. Playback of these camera paths writes the frame buffer contents to a series of image files to the disk for a predefined frame rate, usually 30 FPS. This allows easy demo video creation and reconstruction of camera angles to measure performance or image quality differences when changing shaders and restarting the application.

Besides, a tiled rendering system was integrated that splits up the perspective view frustum into individual smaller parts and renders them at frame buffer-size detail. A series of images is written to the disk for each part of the view frustum. By putting these images together again in an offline tool, one large image can be reconstructed with a resolution larger than what Direct3D 10 allows the frame buffer to be. This is useful for high-detail poster image creation as well as to see extremely small detail on the leaf surfaces, which would otherwise be hidden by texture filtering. Figure 6.9 illustrates a high-detail close-up taken from an output image of 8192x6144.

Fig. 6.9: A close-up view of tiled rendering. High detail remains when zooming in onto a small part of the source image.

6.4 Performance

6.4.1 Frame Rates

The demo application was tested on an Intel Core 2 PC at 2.7 GHz with 2 GB RAM and an NVIDIA GeForce 8800 GTX graphics card with 768 MB

VRAM. The graphics driver version is 180.48. The demo application runs at a resolution of 1024x768 pixels. With one tree in the scene, performance ranges from 70 to 150 FPS depending on the pixel coverage of the tree in the frame buffer. For performance graph recording, the camera is moved forward into the tree continuously. Figure 6.10 illustrates the camera positions for the far and near endpoint of the linear camera path used for performance testing. All performance graphs in this section were recorded using this camera path.

Fig. 6.10: A far-away point of view with about 150 FPS (left) and a viewpoint inside the treetop with 70 FPS (right).

Algorithm Performance

Computations for the animation are performed in the same complexity for each frame, so they do not influence the overall performance depending on tree coverage in the frame buffer. In the current implementation animation is performed three times per frame: First, it is calculated when rendering into the shadow map, a second time when rendering into the depth-first pass and a third time in the scene color pass. This could of course be optimized by using the write-to-vertex-buffer functionality of Direct3D 10 to calculate transformations once per frame and re-use the same buffer for all three passes. Deactivating animation altogether currently brings a small benefit from 80 to about 86 FPS, which demonstrates that it is comparatively cheap to evaluate even if performed three times per frame. Switching between interleaved vertex streams and one individual stream per vertex attribute does not alter performance noticeably.

The complex leaf shader is clearly fillrate bound, so the amount of leaf pixels in the frame buffer directly influences performance. When using a simple Gouraud-shading pixel shader instead the frame rate doubles to a range of 140 to 260 FPS, again depending on the pixel coverage of the tree in

	static	animated	difference
reference	7.14ms (140 FPS)	8.26ms (121 FPS)	1.12ms
unshaded	5.21ms (192 FPS)	5.71ms (175 FPS)	0.50ms
simplified	7.14ms (140 FPS)	7.81ms (128 FPS)	0.67ms
4 trees	16.67ms (60 FPS)	19.23ms (52 FPS)	2.56ms

Tab. 6.1: Framerate comparison and animation-only time in the unshaded, shaded, simplified animation and multiple trees case.

the frame buffer. (Alpha testing is still performed.) Similarly, the frame rate drops from 80 to 50 if the frame buffer resolution is increased to 1920x1080 pixels. Using the depth-first pass has a small payoff when the camera is moved into the treetop and leaf pixels are filling most of the frame buffer, resulting in 70 FPS instead of 60. At far-away viewpoints the depth-first pass has no frame rate benefit, but also does not impact the frame rate negatively. Figure 6.11 illustrates rendering performance for different leaf shader complexities and animation turned on or off.

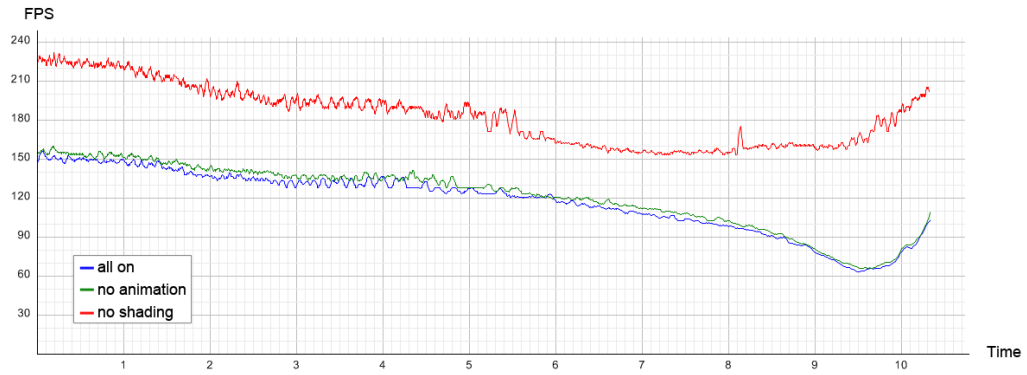

Fig. 6.11: Performance curves when rendering normally (blue), rendering without animation (green) and rendering with a simple Gouraud-shading effect instead of the full leaves shader (red).

Table 6.1 shows the average framerate and milliseconds per frame with and without animation. Also, a simplified animation model is tested, which only displaces vertex positions and omits normal and tangent transformations or length corrections. Simplified animation minimizes the performance impact but still retains a high visual quality if deflections stay within a small scale.

Shadow Mapping Performance

The size of the shadow map is another important performance factor. If a camera angle is chosen with the framerate of 70 FPS for a shadow map size of 4096 pixels per dimension, then it is about 85 FPS for 2048 and 95 FPS for 1024 pixels. This is mostly due to caching problems with large shadow maps, where each branch segment to be rendered has to take a lookup in a different part of the shadow map render target. Changing the size of the tree in the shadow map render target by adjusting the orthographic projection matrix to observe possible fillrate problems when rendering into the shadow map does not alter performance noticeably. Performance curves for different shadow map resolutions can be seen in Figure 6.12.

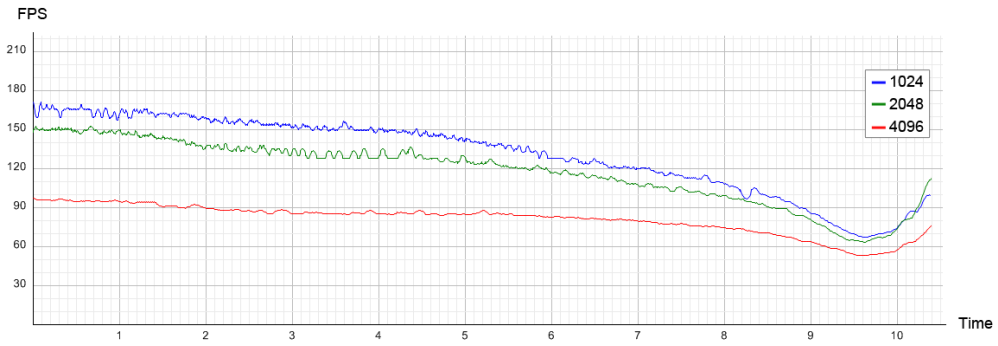

Fig. 6.12: Performance for a 1024*1024 shadow map (blue), a 2048*2048 shadow map (green) and a 4096*4096 shadow map (red).

Multiple Trees

The demo application allows rendering multiple trees aligned in a two-dimensional grid on the XZ plane. Tree positions are perturbed randomly within a grid cell, as are the rotation and size values of each tree to add visual variation. No frustum culling or other techniques are used to cull trees outside the view frustum and no LOD techniques were employed, so the geometry of all trees is transformed on the GPU each frame. This allows predicting the performance impact of multiple trees on the frame rate, which is especially important when considering animation. Figure 6.13 illustrates how a large number of trees limits the performance at the vertex shader stage, resulting in continuous low frame rates. As animation becomes the limiting factor for more than 16 trees, the framerate does not change much for different camera angles.

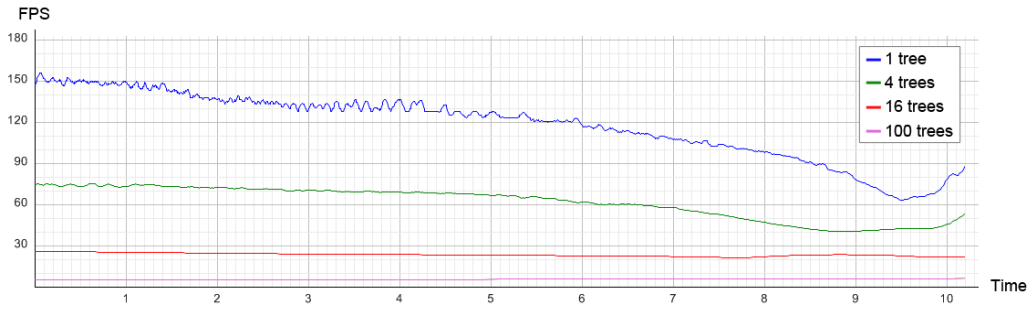

Fig. 6.13: Performance for one tree (blue), 4 trees (green), 16 trees (red) and 100 trees (magenta). As the number of trees increases the frame rate is stabilized by vertex throughput limitations on the GPU.

6.4.2 Shader Statistics

The Direct3D 10 FX compiler is used by Direct3D internally to compile *FX* files containing *HLSL* shader code into a series of assembly instructions which are then uploaded onto the GPU to execute. The stand-alone command-line application of this compiler can be executed by calling "fxc.exe" and is part of the Direct3D 10 SDK. This application is used to measure the number of ALU and TEX instructions in the shaders to allow an estimatation of shader complexity. The November 2008 SDK is used for these calculations.

The FX compiler reports approximately 158 instructions needed for the leaf pixel shader including shadow mapping and the Cook-Torrance specularity model. The vertex shader containing animation uses approximately 259 instructions. Without animation only 42 instructions are needed.

Chapter 7

Conclusion

7.1 Summary

Two algorithms were introduced, one to render convincing images of leaves at high detail, and one to animate complex trees with thousands of branches in real time. Both algorithms are designed to do offline preprocessing in specialized tools and to rely only on vertex and pixel shaders of the GPU at runtime. This makes them easy to integrate into any modern shader-based rendering pipeline. Another benefit of a GPU-based evaluation is that both algorithms scale linearly with the number of pixels to render or vertices to transform. This property makes the techniques suitable for LOD techniques.

The leaf rendering algorithm presented in this thesis takes the surface structure of leaves as well as their optical properties into account, resulting in physically based translucency and subsurface scattering effects. A broad specularity can be observed for the top side of a leaf, while translucency is the dominant feature for the sun-averted side. Photographs and 3D scans of leaves were used to capture the details of a leaf at sub-millimeter accuracy and allow the camera to move within centimeters of a leaf and still retain full detail. Translucency and subsurface scattering are evaluated by transferring the light vector into the "Half Life 2" basis, which allows fast and efficient hemispheric lighting calculations in the pixel shader. Indirect illumination is also part of the algorithm and is precomputed into an ambient occlusion term that also takes light transmittance through leaves into account.

The animation algorithm is based on a stochastic wind model and moves the branches as uncorrelated harmonic oscillators, based on their physical properties as well as where sub-branches and leaves are attached to them. A non-linear bending of branches as they react to incoming wind is possible because it can be represented in a closed form equation and therefore the algorithm is executed in the vertex shader. This decouples the complexity of the animation algorithm from the number of hierarchically connected branches in the model (which would be the case with existing structural el-

ements models and is the limiting factor in those techniques). The same algorithm can be used for a simplified LOD model with 100 branches and for a high-detailed close-up model with 2000 branches without modifications.

Animation data for trees can be created from tree geometry models exported from standard tree generation tools like natFX [6] or XFrog [28] without any further manual processing. A small set of intuitive parameters is used at runtime to shape the appearance of the animation. This makes the process of adding new tree models to an application very fast and efficient, as preprocessing only takes between 5 and 10 minutes for highly detailed tree models with 2000 branches.

A proof-of-concept demo application was created using DirectX 10 and state-of-the-art rendering techniques like High Dynamic Range rendering in combination with various post processing effects were implemented to show that the algorithms presented in this thesis can be integrated into a rendering environment under real-time constraints. The algorithms have to run at least at 60 frames per second while leaving enough GPU resources to calculate the other parts of the pipeline. The demo application was further used to evaluate performance characteristics.

7.2 Further Work

Models to render and animate highly detailed trees and bushes are presented in this thesis, but there are still many possible extensions to the existing models which seem worthwile to explore:

One straight-forward area to go into is the integration of a static or dynamic LOD system into the algorithms, thus proving their efficiency for close-up high-detail geometric models as well as for far-away low-detail billboard cloud representations. Geometry and shader simplification should go hand in hand here. Especially the physically based translucency component of the leaf shader is a property which can be seen even at great distances, so it should be incorporated into all LOD versions of the pixel shader. Other parts like the specularity are also visible at a distance but can be simplified, while parts like normal mapping or shadow mapping could be removed entirely without affecting the resulting image. Smooth blending between different levels of detail will be necessary to avoid LOD popping artifacts. Adaptation of the animation algorithm to fit billboard clouds is of a more complex nature: Multiple levels of branch hierarchy may be encoded into one texture rendered onto a billboard, but a coherent hierarchical animation of individual branches should still be retained. No convincing animation can be realized if only the corned vertices of the quad are transformed.

On the other side, adding new geometric detail to leaves near the camera seems worthwile as well, as leaves are still represented as simple flat quads close-up. Using the geometry shader to calculate a bent leaf form could greatly increase image quality. When modelling large leaves up to a meter long (like the well-known leaves of banana palm trees) the animation system has to be able to convincingly transform the individual vertices of these leaves as well. A simple shearing and rotation along the petiole axis cannot suffice here because wind interaction becomes more complex at this scale. Techniques like the one proposed by Sousa [67] come to mind, which add extra animation information for each leaf vertex to allow a smooth coherent motion.

Further work in the greater area of real-time vegetation rendering could include transferring the leaf model onto grass. Grass has very unique demands when rendered under real-time constraints. LOD and simplification techniques are inevitable because of the extensive amount of grass blades which are visible at once. Overdraw becomes a performance issue as countless blades of grass are rendered in front of each other. Techniques have to be found to avoid filtering artifacts in the frame buffer when rendering grass blades as alpha-textured flat quads, which becomes a problem not only with billboard grass but also with raycasting algorithms.

Many publications deal with simplifying the geometric complexity of grass blades as well as of the rendering algorithm at increasing distance. While individual blades of grass are rendered as geometric models nearby [9] they are rendered as a number of shells atop each other similar to fur rendering at medium distance. A pixel shader using raycasting through few predefined grass slices [32] or simple parallax mapping is used at greater distance. However, these techniques only work for short grass blades and even terrains. Blending between these LODs is another critical factor to ensure a smooth coherent picture when moving the camera in a real-time environment. Of course, shadowing techniqes like shadow mapping may not be necessary to render detailed hard shadows for each grass blade, but smoother indirect lighting approaches like ambient occlusion can be used for self-shadowing of grass blades and the ground between them. Techniques similar to screen-space ambient occlusion [47] could prove valuable here.

Another interesting area of research is the combination of the stochastical per-vertex animation system with a high-level joint system to allow large branches to react on environmental influences. Vertices should be driven both by the per-vertex system as well as by joints calculated on the CPU by a physics simulation. Incorporating correct bending and length correction into this system is one challenge, as is integrating actions such as branches breaking from too strong wind.

List of Figures

List of Tables

Bibliography

[1] Y. Akagi and K. Kitajima. Computer animation of swaying trees based on physical simulation. *Computers and Graphics*, 30(4):529–539, 2006.

[2] Autodesk FBX. `http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=6837478`.

[3] Autodesk Maya. `http://www.autodesk.com/maya`.

[4] Gladimir V.G. Baranoski and Jon G. Rokne. An Algorithmic Reflectance and Transmittance Model for Plant Tissue. *Computer Graphics Forum*, 16(3), 1997.

[5] Gladimir V.G. Baranoski and Jon G. Rokne. Efficiently simulating scattering of light by leaves. *The Visual Computer*, 17(8):491–505, 2001.

[6] natFX. `http://www.bionatics.com/`.

[7] K. E. Bisshopp and D. C. Drucker. Large deflection of cantilever beams. *Quarterly of applied Math*, 3(3):272–275, 1945.

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

[9] Kevin Boulanger, Sumanta Pattanaik, and Kadi Bouatouch. Rendering grass terrains in real-time with dynamic lighting. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 46, New York, NY, USA, 2006. ACM.

[10] Laurent Bousquet, Sophie Lacherade, Stéphane Jacquemoud, and Ismaël Moya. Leaf brdf measurements and model for specular and diffuse components differentiation. *Remote Sensing of Environment*, 98:201–211, 2005.

[11] Yung-Yu Chuang, Dan B Goldman, Ke Colin Zheng, Brian Curless, David H. Salesin, and Richard Szeliski. Animating pictures with stochastic motion textures. *ACM Trans. Graph.*, 24(3):853–860, 2005.

[12] Kim D. Coder. Sway frequency in tree stems. *University Outreach Publication*, FOR00-24, 2000.

[13] Michael F. Cohen, John Wallace, and Pat Hanrahan. *Radiosity and realistic image synthesis*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[14] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, 1982.

[15] Crazybump. `http://www.crazybump.com/`.

[16] Eugene d'Eon, David Luebke, and Eric Enderton. Efficient rendering of human skin. In *Rendering Techniques*, pages 147–157, Grenoble, France, 2007. Eurographics Association.

[17] Julien Diener, Mathieu Rodriguez, Lionel Baboud, and Lionel Reveret. Wind projection basis for real-time animation of trees. *Computer Graphics Forum (Proceedings of Eurographics 2009)*, 28(2), mar 2009. to appear.

[18] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165, New York, NY, USA, 2006. ACM.

[19] Craig Donner and Henrik Wann Jensen. Light diffusion in multi-layered translucent materials. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1032–1039, New York, NY, USA, 2005. ACM Press.

[20] Cass W. Everitt and Mark J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. *CoRR*, cs.GR/0301002, 2003.

[21] Randima Fernando. Percentage-closer soft shadows. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 35, New York, NY, USA, 2005. ACM.

[22] Oliver Franzke and Oliver Deussen. Rendering plant leaves faithfully. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1, New York, NY, USA, 2003. ACM Press.

[23] Barry D. Ganapol, Lee F. Johnson, Philip D. Hammer, Christine A. Hlavka, and David L. Peterson. LEAFMOD: A new within-leaf radiative transfer model. *Remote Sensing of Environment*, 63:182–193, 1998.

[24] Andrew Gardner, Chris Tchou, Tim Hawkins, and Paul Debevec. Linear light source reflectometry. *ACM Trans. Graph.*, 22(3):749–758, 2003.

[25] Geomagic. `http://www.geomagic.com/`.

[26] Crytek GmbH. Crysis.

[27] Yves M. Govaerts, Stéphane Jacquemoud, Michel M. Verstraete, , and Susan L. Ustin. Threedimensional radiation transfer modeling in a dycotyledon leaf. *Applied Optics*, 35(33):6585–6598, 1996.

[28] XFrog. `http://www.xfrogdownloads.com/`.

[29] Ralf Habel, Alexander Kusternig, and Michael Wimmer. Physically based real-time translucency for leaves. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)*, pages 253–263. Eurographics, Eurographics Association, June 2007.

[30] Ralf Habel, Alexander Kusternig, and Michael Wimmer. Physically guided animation of trees. In *Proceedings of the Eurographics 2009*, April 2009. to appear.

[31] Ralf Habel, Bogdan Mustata, and Michael Wimmer. Efficient spherical harmonics lighting with the preetham skylight model. In Katerina Mania and Erik Reinhard, editors, *Eurographics 2008 - Short Papers*, pages 119–122. Eurographics Association, April 2008.

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

[33] William Van Haevre, Fabian Di Fiore, Philippe Bekaert, and Frank Van Reeth. A ray density estimation approach to take into account environment illumination in plant growth simulation. In *SCCG '04: Proceedings of the 20th spring conference on Computer graphics*, pages 121–131, New York, NY, USA, 2004. ACM.

[34] Pat Hanrahan and Wolfgang Krueger. Reflection from layered surfaces due to subsurface scattering. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 165–174, New York, NY, USA, 1993. ACM Press.

[35] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518, New York, NY, USA, 2001. ACM Press.

[36] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed Shape Representation with Parallax Mapping. In *International Conference on Artifical Reality and Telexistance 2001*, 2001.

[37] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93*, pages 145–153, 1993.

[38] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Second Edition.* Charles River Media, Inc., Rockland, MA, USA, 2003.

[39] Xinguo Liu, Peter-Pike Sloan, Heung-Yeung Shum, and John Snyder. All-Frequency Precomputed Radiance Transfer for Glossy Objects. *Proceedings Eurographics Symposium on Rendering*, 15:337–344, 2004.

[40] Tobias Martin and Tiow-Seng Tan. Anti-aliasing and continuity with trapezoidal shadow maps. In *Eurographics Symposium on Rendering Proceedings*, pages 153–160, 2004.

[41] Nelson L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4:109–117, 1988.

[42] Gary McTaggart. Half-Life 2/Valve Source Shading. Technical report, Valve Corporation, 2004.

[43] Tom Mertens, Jan Kautz, Philippe Bekaert, Frank Van Reeth, and Hans-Peter Seidel. Efficient rendering of local subsurface scattering. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 51, Washington, DC, USA, 2003. IEEE Computer Society.

[44] Jason Mitchell. Real-time 3d scene post-processing. Talk, GDC, 2003.

[45] Jason Mitchell. Light shafts. Talk, GDC, 2004.

[46] Kenny Mitchell. Volumetric light scattering as a post-process. In *GPU Gems 3*, pages 275–285. Charles River Media, 2007.

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

[48] John R. Moore and Douglas A. Maguire. Natural sway frequencies and damping ratios of trees: concepts, review and synthesis of previous studies. *Trees*, 3(18):195–203, 2004.

[49] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. *Geometrical considerations and nomenclature for reflectance*. Jones and Bartlett Publishers, Inc., USA, 1977.

[50] Tomoyuki Nishita, Yoshinori Dobashi, and Eihachiro Nakamae. Display of clouds taking into account multiple anisotropic scattering and sky light. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 379–386, New York, NY, USA, 1996. ACM.

[51] NVIDIA Plug-ins for Adobe Photoshop. `http://developer.nvidia.com/object/photoshop_dds_plugins.html`.

[52] Shin Ota, Machiko Tamura, Tadahiro Fujimoto, Kazunobu Muraoka, and Norishige Chiba. A hybrid method for real-time animation of trees swaying in wind fields. *The Visual Computer*, 20:613–623(11)], dec 2004".

[53] R. Perez, J.R. Seals, and J Michalsky. An all weather model for sky luminance distribution, 1993.

[54] Ken Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1985. ACM Press.

[55] Matt Pharr. Ambient occlusion. Talk, GDC, 2004.

[56] Arcot J. Preetham, Peter Shirley, and Brian Smits. A practical analytic model for daylight. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 91–100, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[57] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[58] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87 Proceedings*, pages 283–291, New York, NY, USA, 1987. ACM Press.

[59] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Trans. Graph.*, 21(3):267–276, 2002.

[60] Tatsumi Sakaguchi and Jun Ohya. Modeling and animation of botanical trees for interactive virtual environments. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 139–146, New York, NY, USA, 1999. ACM.

[61] Thorsten Scheuermann. Advanced depth of field. Talk, GDC, 2004.

[62] Mikio Shinya and Alain Fournier. Stochastic motion-motion under the influence of wind. *Comput. Graph. Forum*, 11(3):119–128, 1992.

[63] Peter Shirley and Kenneth Chiu. A Low Distortion Map Between Disk and Square. *Journal of Graphics Tools*, 2(3):45–52, 1997.

[64] Peter-Pike Sloan. Normal mapping for precomputed radiance transfer. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 23–26, New York, NY, USA, 2006. ACM Press.

[65] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02 Proceedings*, pages 527–536, New York, NY, USA, 2002. ACM Press.

[66] Peter-Pike Sloan, Ben Luna, and John Snyder. Local, deformable precomputed radiance transfer. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1216–1224, New York, NY, USA, 2005. ACM.

[67] Tiago Sousa. Vegetation procedural animation and shading in crysis. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 16, pages 373–386. Addison Wesley, July 2007.

[68] Jos Stam. *Multi-scale stochastic modelling of complex natural phenomena*. PhD thesis, University of Toronto, Toronto, Ont., Canada, Canada, 1995.

[69] Jos Stam. Stochastic dynamics: Simulating the effects of turbulence on flexible structures. *Computer Graphics Forum*, 16(3):C159–C164, 1997.

[70] Marc Stamminger and George Drettakis. Perspective Shadow Maps. In John Hughes, editor, *SIGGRAPH 2002 Conference Proceedings*, Annual Conference Series, pages 557–562. ACM Press/ ACM SIGGRAPH, 2002.

[71] Katsumi Tadamura, Xueying Qin, Guofang Jiao, and Eihachiro Nakamae. Rendering optimal solar shadows using plural sunlight depth buffers. In *Computer Graphics International 1999*, page 166, 1999.

[72] Stephen Timoshenko, Donavan YOUNG, and WEAWER, Williams, Jr. *Vibration problems in engineering*. New-York : 1974, 1974.

[73] E.A. Walter-Shea, J.M. Norman, and B.L. Blad. Leaf bidirectional reflectance and transmittance in corn and soybean. *Remote Sensing of Environment*, 29:161–174, 1989.

[74] Lifeng Wang, Wenle Wang, Julie Dorsey, Xu Yang, Baining Guo, and Heung-Yeung Shum. Real-time rendering of plant leaves. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 712–719, New York, NY, USA, 2005. ACM Press.

[75] Rui Wang, John Tran, and David Luebke. All-frequency interactive relighting of translucent objects with single and multiple scattering. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1202–1207, New York, NY, USA, 2005. ACM Press.

[76] Fabian Di Fiore William Van Haevre and Frank Van Reeth. Physically-based driven tree animations. *Eurographics Workshop on Natural Phenomena*, pages 75–82, 2006.

[77] Michael Wimmer and Daniel Scherzer. Robust Shadow Mapping with Light Space Perspective Shadow Maps. In Wolfgang Engel, editor, *ShaderX 4 – Advanced Rendering Techniques*, volume 4 of *ShaderX*. Charles River Media, March 2006.

[78] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. Parallel-split shadow maps for large-scale virtual environments. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality*

continuum and its applications, pages 311–318, New York, NY, USA, 2006. ACM.

Acknowledgements

I would like to send my most sincere thanks to the following people:

Georg Semanek and my brother *Michael* for proof-reading the thesis and listening to my ongoing ramblings about everything related to it.

The *Institute of Computer Graphics* for providing the advisory support for my thesis, and especially *Ralf Habel* for putting the demo application to good use in a number of publications.

The *Faculty of Computer Science* for granting me a scholarship to work on this thesis, and especially *Michael Wimmer*, *Ralf Habel* and *Werner Purgathofer* for supporting my application for the scholarship.

And finally, *my family*, who endured two long years of "I'll be done soon."