Master 2 de Recherche en Informatique
IFSIC, Université de Rennes I

Bibliography report

Real-Time Realistic Rendering of Grass with Lighting

Kévin Boulanger

January 2005

Supervised by:

Sumanta Pattanaik, UCF (University of Central Florida)
Kadi Bouatouch, IRISA, SIAMES project

Contents

1 Introduction

Nowadays, computer generated images are getting more and more realistic. Indeed, the targeted goal is to create images that are similar to what one can see in real life with his eyes. However, we come up against an obstacle: complexity. Scenes from real life contain a huge number of small details which are hard to model, take a lot of time to render and require a huge amount of memory, unavailable in current computers. This complexity mainly comes from geometry and lighting. The geometric complexity is due to a high number of grass blades for example, a huge number of primitives such as triangles are needed to accurately model grass. Lighting computations complexity is due to the multiple reflections of light over the scene objects with complex materials.

Overcoming this complexity has been a challenging problem for many years. Grass rendering belongs to this kind of problem. Some algorithms have been developed to render grass in real-time with approximations, others allow to get high quality results such as raytracing. Even in the last case, approximations have to be used to obtain acceptable processing times. Moreover, grass rendering is important because many people need to use it, for instance in flight simulators, soccer video games, walk-through inside virtual nature environments, etc.

Some techniques have been developed in parallel to render volumetric objects. We can use this kind of technique to render grass with full parallax. Realistic modeling of materials, with Bidirectional Texture Functions for example, can be used in our case to render distant grass with realistic light interactions.

Our goal is to render grass with the highest fidelity, as fast as possible. Real-time is our goal, however some of the existing algorithms can meet this goal and others not. We want to choose the fastest ones which offer the possibility of quality rendering.

This report is structured as follows. Firstly, we will talk about grass rendering in general: characteristics of grass have to be studied to find out bottlenecks of the existing algorithms and of our future implementation. Then, some existing grass rendering methods will be presented, with their pros and cons. Finally, we will present our proposed solution. Some words will be said about the software architecture necessary for the development of a grass rendering library.

2 Grass rendering characteristics

In this section, we talk about grass rendering in general. The fact of giving characteristics of grass allows us to find out the pros and cons of some of the existing rendering methods.

We have classified the different characteristics and parameters into four categories which are presented hereafter.

2.1 Natural characteristics

The first characteristic is the presence of different *grass species*: one kind of grass can be displayed, or several ones at the same time. If only one is displayed, some repetition will be observed. With several ones, the data structures management is harder.

Grass can be short, medium or long. The dynamic grass behavior changes in function of its *length*: long grass is more influenced by wind than short grass. Length can also vary with the kind of underlying terrain: small in places where pedestrians often walk and medium in other places. Another example of short grass is a soccer stadium. Long grass can be found in a meadow.

Density is the number of grass blades per unit of surface. If this number increases, the grass hides a larger surface of ground and looks more realistic but needs more computation power for rendering. The variation of density is also a factor of realism: real grass is in general more dense where high amount of water is present beneath ground. This distribution is not homogeneous in real life. These variations are modeled with additional parameters which can be hard to manage.

Grass grows on several *kinds of grounds*: soil, concrete, swamp, etc. This ground diversity influences the growing speed and the density.

In real life, grass is often associated with *other plant species* such as flowers, bushes, trees, etc. A scene containing grass will look more realistic if such a vegetation is rendered at the same time. Stones, rocks, insects can also be added.

Grass can be wet or dry. *Dampness* influences light reflection: more light is reflected in places containing more water due to dew or rain.

Finally, grass can have another *color* than green. In a soccer stadium, lines and circles are painted onto the grass to mark the game zone bounds. To render it, an additional color map must be supplied and can affect the rendering time.

2.2 Speed and quality of rendering

The *rendering speed* has to be high to allow for an immersion sensation into 3D scene. However, highly complex scenes such as vast surfaces of grass are hugely processing consuming and hence very difficult to render in real-time. Many optimizations and approximations are necessary to speed up the rendering.

When high *quality* is needed, real-time rendering is usually very difficult to obtain. Classical rasterization methods do not allow for high quality rendering. Instead, raytracing or other derived methods allow for better lighting and shadowing. Unfortunately, they require several hours of computation.

GPUs (Graphics Processing Units) can be employed to highly speed up rendering. A GPU can perform a great number of floating point operations per second, more than a CPU *(Central Processing Unit)*. Nowadays, programmable GPUs allow for very complex calculations such as high quality light reflections. It is very difficult to use a GPU for raytracing because it is more suited to rasterization algorithms.

In real life, grass moves because of wind. Grass can be *animated* using physical simulation. However, it requires a high simulation time to the detriment of rendering time. Rendering quality has to be lost to animate grass in real-time.

Finally, grass is said to have *full parallax* if it can be observed from any point of view without bad depth effect. Many grass rendering algorithms do not offer full parallax. For instance, some methods need rendering from walker's height to avoid the bad depth effect.

2.3 Data structures

Data representing grass can be *generated* in two different ways: procedurally or from measures of real grass blades.

The amount of needed *memory* is an important criterion: data structures needed to store grass blade descriptions can be demanding in terms of memory resource. Some algorithms are less memory intensive, for instance those using image-based rendering.

The amount of needed *hard disk space* is another problem: to store measured grass samples, a large disk space is required, particularly for reflectance parameters. Volumetric textures need also a huge amount of data. Conversely, when grass description is procedurally generated, a small hard disk space is required, only the generators parameters have to be stored.

Finally, *levels of detail* have to be managed: a grass rendering algorithm can apply to different levels of detail. However, a rendering method is usually suitable only for a specific scale. For example, only points of view near grass can use geometry-based rendering, otherwise too many triangles have too be rendered.

2.4 Lighting conditions

To obtain realistic rendering of animated grass, *dynamic lighting* has to be used but it is really time consuming. Lighting conditions can be precalculated. In this case, lighting is static and can be sufficient when the animation of grass blades consists of small movements and when light sources do not move (during a short period of time, the sun can be considered as static). The easiest and fastest method consists in rendering grass without lighting. For a scene with very bright sun rays, the result quality can be acceptable.

Light can be decomposed into *ambient*, *diffuse* and *specular components*. Ambient lighting for natural scene is mainly provided by light diffusion inside the atmosphere. In addition, a blue shift happens because the atmosphere absorbs the red part of the spectrum more than the blue one. Diffuse lighting is due to the diffusion of light beams over the surfaces of the scene. This component

is the main part of the light we can see in real life. The specular lighting component corresponds to direct reflections of light over the surfaces. It is view dependent and so cannot be precalculated.

By *day* or by *night*, lighting conditions are different. By night, diffuse lighting can be ignored and only ambient is used. A dark blue ambient lighting would create a realistic scene with few computations. Even with rendering of specularities due to the moon light, frame rate can be similar to that entailed when rendering scenes by day.

Finally, *shadowing* is needed to obtain realistic images. There are different kinds of shadow: self-shadowing (a part of a grass blade casts a shadow onto another part of this blade), shadows cast by some grass blades onto other blades, and shadows due to the environment (trees for instance). Shadows management is very time-consuming. It is a problem for real-time rendering, in particular when rendering soft shadows. With today's 3D cards, hard shadows are preferred with complex scenes such as grass surfaces.

3 Existing grass rendering methods

Grass rendering is a domain which has been already studied. Some of the papers we present here focuses on grass rendering. Others present different methods to render natural objects such as trees. We can use variations of their methods for rendering of grass. We present also methods that will allow us to develop realistically lighted grass for our implementation.

Grass can be seen from different points of view: near the ground, at walker's height, from the sky, etc. That means grass blades can have large projections on screen or a single pixel. The different kinds of algorithms we present here are on the whole dedicated to an range of view distances. For each method, we give their possible use.

3.1 Geometry

When 3D objects need to be rendered in real-time, we firstly think about classic rasterization algorithms with primitives such as triangles. Geometry-based methods allows for precise representations of objects, animated if desired, with textures and lighting. However, the processing power needed to render many thousands of grass blades in real-time is nowadays unavailable. Rendering lots of triangles for grass blades which have a size of a pixel is useless. This situation occurs when the camera is far from the grass. Consequently, geometry is required only for grass standing near the camera.

Using geometry allows for an easier management of different grass species, with variable length and density.

3.1.1 Manual modeling

Nowadays, numerous modeling software exist. Modeling of individual grass blades or clumps of them is possible but it is very tedious for complete natural scenes. When repetition of a simple group of grass blades is used, one can see the patterns when rendering. Consequently, this kind of modeling should be used in special cases, for broken grass blades for example.

3.1.2 Procedural modeling

Procedural modeling is more fitted to geometric grass rendering. In this case, the grass designer has very few parameters to modify to obtain very detailed surfaces of grass. Moreover, data to be stored require only a small memory space. Some existing methods are presented hereafter.

In [Ree83] and [RB85], Reeves presents particle systems. They consist of clouds of simple primitives which are animated with procedural algorithms. Particle systems can be used for special effects: fire, waterfall, firework, etc. Using gravity simulation, parabolic trajectories are followed by the particles of an explosive system [Ree83]: particles have an initial speed toward a random direction when created by an emitter. The generated coordinates, to be assigned to the particles, are used to describe grass blade shapes: the initial time t_0 represents the root of the blade, while the final time t_f the end. An example is shown on figure 1*(a)*.

Another possible method to determine the coordinates of the grass blade vertices is the use of cubic Hermite curves. They are presented in [AMH02a]. These curves are defined by the coordinates

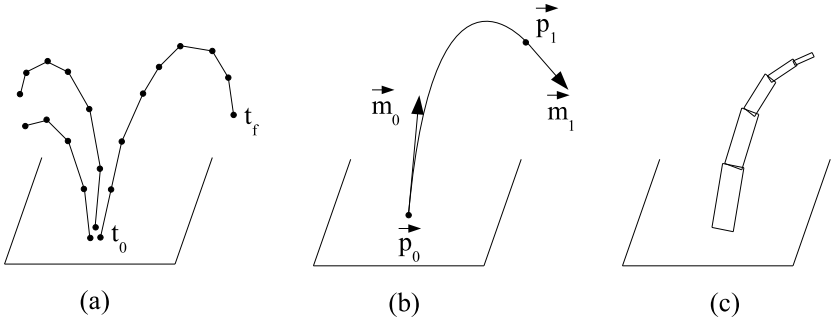

Figure 1: Grass blades using geometry. *(a)* Using a cubic Hermite curve. *(b)* Using trajectories of particles. The dots represent the generated coordinates. *(c)* Consecutive billboards to display a grass blade.

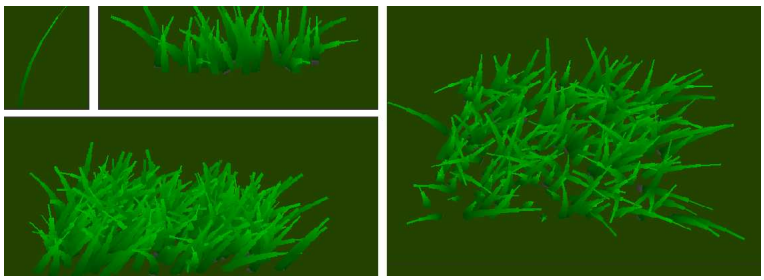

Figure 2: Grass blades with 3D geometry. Top left: a single blade of grass. Top middle: blades located in a slice from a patch. Right: Patch of grass seen from above. Left bottom: the same patch seen from walker's height. Images from [PC01].

$(\vec{p_0}, \vec{p_1})$ and the tangents $(\vec{m_0}, \vec{m_1})$ of their ending points. This case is illustrated on figure 1*(b)*. The coordinates of the points $\vec{p}(t)$ along the curve are, for $t \in [0, 1]$:

$$\vec{p}(t) = (2t^3 - 3t^2 + 1)\vec{p_0} + (t^3 - 2t^2 + t)\vec{m_0} + (t^3 - t^2)\vec{m_1} + (-2t^3 + 3t^2)\vec{p_1}$$

This equation can be easily determined when a cubic polynomial expression has to be used for $\vec{p}(t)$ and when the following hypothesis are defined:

$$\vec{p}(0) = \vec{p_0}, \quad \vec{p}(1) = \vec{p_1}, \quad \frac{\partial \vec{p}(t)}{\partial t}(0) = \vec{m_0}, \quad \frac{\partial \vec{p}(t)}{\partial t}(1) = \vec{m_1}$$

Several possibilities exist to render grass blades. When using particles systems, Reeves proposes in [RB85] to render particles using anti-aliased lines. When using consecutive steps of a particle simulation, connected anti-aliased lines form a blade of grass. This method is also presented by Deussen in [DCSD] for rendering plants constituted of thin and long structures. These structure constraints are applicable to grass blades.

Another method, proposed in [GPR+], makes use of billboards. For instance, 5 billboards allows for efficient rendering of a grass blade (as on figures 1*(c)* and 2).

Distributing blades of grass over the ground

Grass blades has not to be regularly distributed over the ground, otherwise patterns will be visible. Stochastic distribution solves this problem but has to be studied. If dense clumps of grass are present, blades will have many unnatural intersections.

Deussen [DHL+98] simulates ecosystems to distribute plants over the ground. He coarsely represents plants as circles. These latter define an *ecological neighborhood*. The radius of a circle represents the area where the corresponding plant interacts with its neighbors. Biological rules govern the interactions between the intersecting circles. An example of ecosystem simulation result is given on figure 3. Water presence under ground influences plant distribution. An example of such a distribution is given on figure 4.

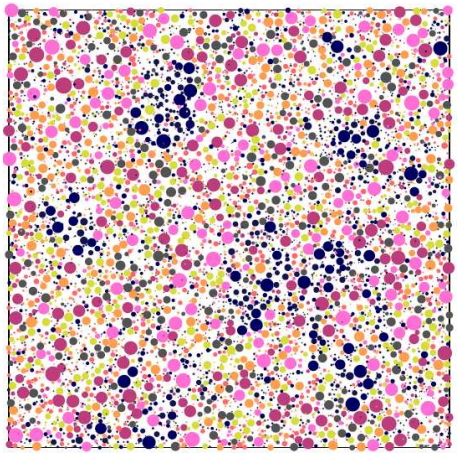

Figure 3: Simulated distribution of eight plant species. Colors indicate plant species. Blue circles stand for plants with a preference for wet areas. Image from [DHL+98]

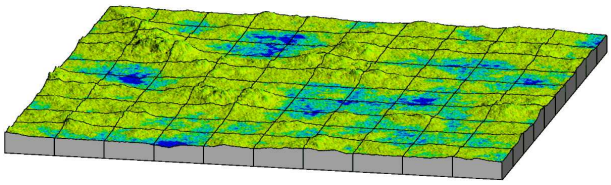

Figure 4: Sample terrain with water density ranging from high (blue) to low (yellow). Image from [DHL+98]

Grass distribution can be computed using a simplified version of the Deussen's ecosystem simulation [DHL+98]. Water distribution has a great impact on grass density, while behavior due to the presence of various grass species has not to be necessarily managed in case of grass rendering.

3.1.3 Lighting geometry

Rendering with geometry (triangles in particular) has been studied for many years, so many lighting algorithms have been developed. Lighting models used in current Graphics Processing Units (GPU) use simple reflectance models (with ad hoc ambient, diffuse and specular components computations). To improve these models, more complex *Bidirectional Reflectance Distribution Functions (BRDF)* based on more accurate physical representations can be used [AMH02b]. They consist in simulating light reflection at a point of a surface using a function depending on the local micro-geometry. A BRDF is a 4D function $f_r(\vec{\omega_i}, \vec{\omega_r})$ where $\vec{\omega_i}$ is the incident light flux direction and $\vec{\omega_r}$ the reflected flux direction (figure 5). These directions are often expressed with spherical coordinates (ϕ_i, θ_i) and

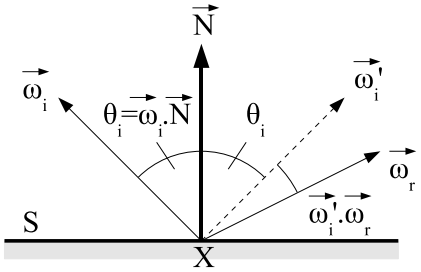

Figure 5: Vectors used by the Phong's BRDF model. Incident light from direction $\vec{\omega_i}$ is reflected on point X of surface S to the direction $\vec{\omega_r}$. $\vec{N}$ is the normal to the surface and $\vec{\omega_i'}$ the ideal reflection direction.

(ϕ_r, θ_r). When the reflection direction $\vec{\omega_r}$ is fixed, the BRDF becomes a 2D function, depending only on $\vec{\omega_i}$. This function can be represented in spherical coordinates as on figure 6. In this case, the BRDF value represents the contribution of incident light of a given direction $\vec{\omega_i}$. An example of well-known BRDF is the *Phong's model*, illustrated on figure 6*(a)* and its equation is:

$$f_r(\vec{\omega_i}, \vec{\omega_r}) = \rho_d + \rho_s(\vec{\omega_i'}.\vec{\omega_r})^n$$

where ρ_d and ρ_s are the diffuse and specular reflectance coefficients, $\vec{\omega_i'} = 2(\vec{N} \cdot \vec{\omega_i})\vec{N} - \vec{\omega_i}$ is the ideal reflection vector, n is the shininess (high for shiny surfaces). Vectors must have an unit length to obtain correct results.

The *radiance equation* that gives the reflected radiance $L_r(\vec{\omega_r})$ is obtained by integrating the product of the BRDF $f_r(\vec{\omega_i}, \vec{\omega_r})$ and the irradiance dA:

$$\begin{aligned} L_r(\vec{\omega_r}) &= L_e(\vec{\omega_r}) + \int_\Omega f_r(\vec{\omega_i}, \vec{\omega_r})\ dA = L_e(\phi_r, \theta_r) + \int_0^{2\pi} f_r(\phi_i, \theta_i, \phi_r, \theta_r) L_i(\phi_i, \theta_i) \cos\theta_i\ d\Omega_i \\ &= L_e(\phi_r, \theta_r) + \int_{\phi_i=0}^{2\pi} \int_{\theta_i=0}^{\pi/2} f_r(\phi_i, \theta_i, \phi_r, \theta_r) L_i(\phi_i, \theta_i) \cos\theta_i \sin\theta_i\ d\theta_i\ d\phi_i \end{aligned}$$

where L_r is the reflected radiance, dA is the *irradiance* (the power impinging onto the surface), L_i is the incident radiance, L_e is the radiance due to self-emission, $d\Omega_i$ is an unit solid angle around an incident direction $\vec{\omega_i}$. The radiance is in fact a *dot product* between $f_r(\phi_i, \theta_i, \phi_r, \theta_r)\cos\theta_i$ and $L_i(\phi_i, \theta_i)$. When implementing the radiance equation, the $\cos\theta_i$ term should be inserted into $f_r(\phi_i, \theta_i, \phi_r, \theta_r)$ and $L_i(\phi_i, \theta_i)$ can be represented by a texture called *environment map*. Using the radiance equation allows to compute the reflected flux at a point of a surface using contribution of the whole environment.

A BRDF is either represented by measured data or by a model fitting these data. As presented in [MMS+], using an analytic model from fitted sampled data allows for rendering in real-time. An example of such a model is *Lafortune lobes* as explained in section 3.2.3.

We have just given the base theory of BRDFs. Application to geometry-based grass rendering is interesting. Based on observations of real life, grass seems to have an anisotropic light reflection model (dependent of the incident light azimuth angle ϕ_i). Kajiya et al. [KK89] define a model for hair rendering which is also applicable to grass. This model uses a surface comparable to a thin cylinder. The equation used is:

$$f_r(\vec{T}, \vec{\omega_i}, \vec{\omega_r}) = \rho_d \sin(\vec{T} \cdot \vec{\omega_i}) + \rho_s \left[(\vec{T} \cdot \vec{\omega_i})(\vec{T} \cdot \vec{\omega_r}) + \sin(\vec{T} \cdot \vec{\omega_i}) \sin(\vec{T} \cdot \vec{\omega_r})\right]^n$$

where $\vec{T}$ is the tangent vector (along a hair). The model is defined by three constants: ρ_d, ρ_s and n which are respectively the diffuse reflection coefficient, the specular reflection coefficient and the Phong exponent. The first two parameters specify the surface intrinsic colors and the last represents the width of the specular lobe. This model is defined using the *Lambert's model* ($f_r(\vec{\omega_i}, \vec{\omega_r}) = \rho_d$) integrated along the circumference of a half cylinder for the diffuse component. The specular component is defined using an ad hoc model, close to a Phong's one, modified to approximate some diffraction.

With current hardware, using a high-level shading language such as *OpenGL Shading Language (GLSL)*, the possibility of implementing complex lighting models for real-time applications exists. The model equations can be directly written in a *fragment shader*, a small program used by current 3D cards for per-fragment computations. To speed up these latter (reducing the quality of the rendering), lighting computations can be transferred to *vertex shaders* that compute the lighting model for each vertex instead of each fragment.

Realism can be further improved using *subsurface scattering*. BRDFs do not simulate light scattering below a surface. In real life, grass blades are translucent. An algorithm, described by Green in [Gre04], approximates subsurface scattering in real-time using *depth maps*. These maps contain the distances from a light source to the scene objects and are used to compute an approximation of the objects thickness. The absorption is considered as varying linearly with the distance the light travels inside the objects.

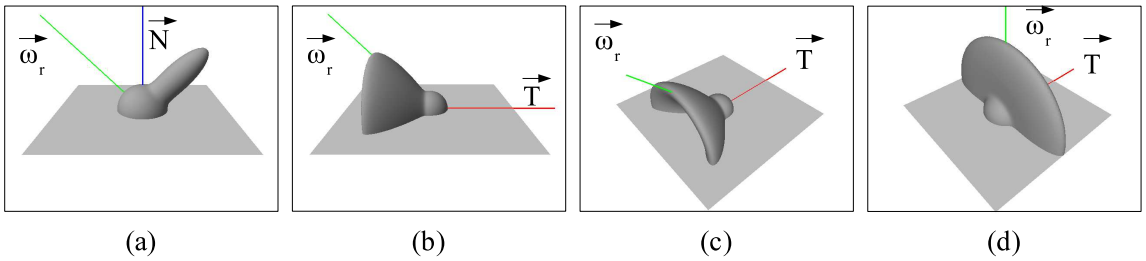

Figure 6: Examples of BRDFs. *(a)* Phong's model (isotropic). *(b), (c), (d)* Kajiya's model (anisotropic).

In section 3.1.2, we have presented a method to render grass easily: antialiased lines where coordinates are defined using particle systems. Color variations of these lines can simulate lighting. In [RB85], Reeves and Blau proposed methods to approximate light scattering inside clumps of grass defined using particle systems. At the top of the grass blades, ambient and diffuse components are at their maximum. When going down to the ground, these components decrease. The diffuse component contribution decreases faster than the ambient component.

To simulate shadows casting on grass, Reeves and Blau propose in [RB85] to use *shadow masks* which contain an orthographic greyscale image of the environment, in particular trees. This image can be projected over the ground, modulating then the grass blades lines colors.

3.1.4 Animating geometry

In real life, grass is not static. Its dynamic behavior is mainly due to the wind and other events such as crushing by walkers or rolling stones, and can be simulated with approximative procedural methods or physically-based models. To obtain real-time rendering, procedural approaches are generally used. Due to the visual complexity of grass, exact physically-based animation is very time-consuming.

To obtain efficient animation of grass, wind effect in particular, consequences on grass are modeled but not the causes. In this case, the animation is procedurally defined. For example, in [Pel04], Pelzer proposes to use trigonometric functions to approximate the swaying of grass blades. In [PC01], Perbet and Cani use procedural animation primitives: slight breeze, gust of wind, whirlwind, blast of air. These primitives send parameters to the grass blades to indicate how they must be bent. Bent geometric blades are pre-computed using physically-based simulation. A value is used for real-time animation to indicate to grass blades in what position they must be.

3.2 Image-Based Rendering (IBR)

Image-Based Rendering uses one or several pre-computed images to represent a 3D object. It is often used when geometry is too complex to be efficiently rendered. IBR allows to use primitives which cover the screen's pixels as efficiently as possible. However, this representation is only suited to very small objects, or for objects that stand far from the view point. Many methods exist and we present some of them in the following sections.

3.2.1 Simple textures

Simple texturing may be used for very faraway grass. It consists in mapping one or several 2D textures onto the quadrilaterals defining the terrain covered with grass. As pointed out in [PC01], the drawback of this method is that lighting and animation are not realistic.

It is difficult to represent various species of grass of various densities at the same time because many different textures need to be used. The represented grass can include long blades when the distance from the viewer is long. With a shorter grass, the textured surfaces can be seen from a close point of view. With this representation, it is easy to paint lines on the grass in a soccer stadium: the lines are part of the textures.

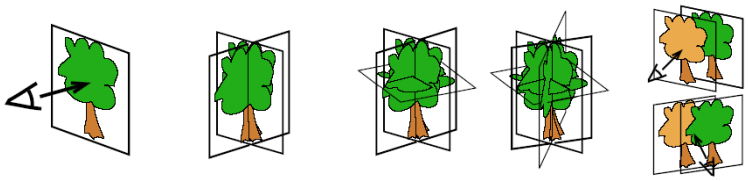

Figure 7: Billboards techniques. From left to right: simple billboard, cross billboard, 3-cross billboard, 4-cross billboard, popping of billboards when the camera is moving. Images from [DN04].

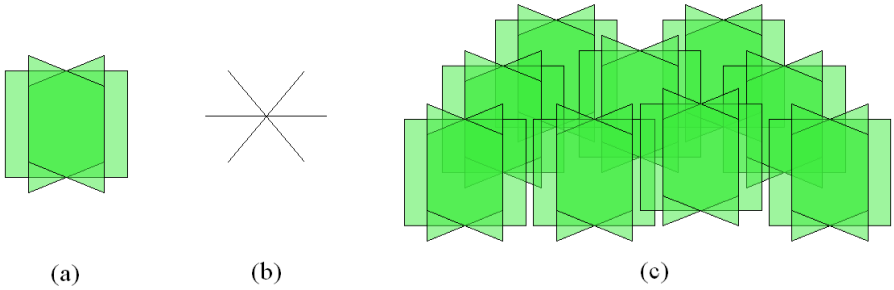

Figure 8: Cross billboards for grass. *(a)* A single cross billboard. Each quadrilateral is mapped with a grass 2D texture. *(b)* The cross billboard seen from above. *(c)* A set of cross billboards randomly positioned over the ground.

3.2.2 Billboards

Billboards are triangles or quadrilaterals covered by a semi-transparent texture. They allow to render objects with a non-triangular or non-quadrilateral border, such as trees as seen from far. Their rendering is more efficient than that of classical geometry since they can contain lots of objects inside a single texture. Likewise, that is useful for very complex objects such as grass blades.

Simple billboards

Simple billboards are well suited to the representation of single objects such as trees. They were used in old video games due to their low rendering cost. As shown on the left of figure 7, a single quadrilateral is rendered with a semi-transparent texture which is always facing the camera. A verticality constraint is often added to keep the represented object vertical, particularly trees. When two billboards are close, popping artifacts appear when the camera is moving. This effect is illustrated on the right of figure 7. So, simple billboards are adapted only to sparse objects. Moreover, this method shows no parallax when the camera is moving, decreasing the realism of the rendered images.

Cross billboards

To show more parallax when the camera is moving, several quadrilaterals have to be rendered for static billboards. For example, these quadrilaterals may be placed as in the second, third and fourth images of figure 7.

This kind of configuration is retained by Pelzer in [Pel04]. He uses three vertical quadrilaterals creating a star if seen from above. He renders a large number of such cross billboards with random positions over the ground. Therefore numerous intersections occur, preventing the creation of repetitive patterns. This method is illustrated on figure 8.

Aligned layers of billboards

Aligned layers of billboards is a simpler method that consists in placing layers of vertical strips in the same orientation. It is illustrated on figure 9 and presented in [PC01]. To avoid appearance of the structure when the camera is moving, orthogonal sets of layers are defined and the correct orientation is chosen when rendering. Even if a stochastic deformation of the billboards is able to

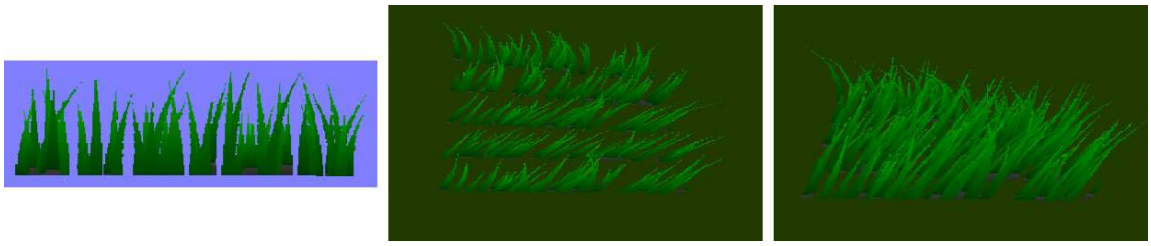

Figure 9: Grass blades with aligned layers of billboards. Left: polygon strip mapped with a RGBA texture. Middle: layers of billboards seen from above. Right: same billboards seen from walker's height. Images from [PC01].

reduce pattern repetition, this method introduces many artifacts, in particular if grass is viewed from above (middle of figure 9).

3.2.3 Lighting with IBR

Simple textures

Rendering quadrilaterals covered with 2D textures is a very fast operation. So, more complex lighting models could be used without affecting interactivity.

To obtain realistic materials, a promising representation is the *Bidirectional Texture Function* (BTF) also known as *Spatial Bidirectional Reflectance Distribution Function* (SBRDF). This function describes the meso-structures of a material, between the macro-structures modeled with triangles and micro-structures modeled with BRDFs (*Bidirectional Reflectance Distribution Functions* presented in section 3.1.3). A BTF is a 6-dimensional function $f_r(X, \vec{\omega_i}, \vec{\omega_r})$ where X is a point of a surface, $\vec{\omega_i}$ the direction of the *incident light flux* and $\vec{\omega_r}$ the direction of the *reflected flux* (figure 5). A sampled BTF can be stored either as a set of discrete textures for each light and reflection directions, or as a set of *apparent BRDFs*, one for each point X. They are called *apparent* because the underlying geometry scale is larger than that of a classical BRDF. Therefore, BTFs enable self-shadowing and management of self-occlusion. In fact the content of the BRDF is multiplied by a visibility function $V(X, \vec{\omega_i}) \in [0, 1]$ which is equal to 1 when light is directly visible from the point X of the surface in the direction $\vec{\omega_i}$ and 0 when it is totally hidden by a neighbor obstacle. In their state-of-the-art publication [MMS+], Müller et al. describe how to acquire BTFs from real life surfaces, compress them using various factorization methods and render surfaces using materials defined with BTFs. When rendering directly observable surfaces using rasterization methods, the view vector is considered as the reflection vector: a vector from a point of the surface to the camera.

BTFs can be used to represent a patch of grass from different view and light directions. Using graphics hardware, patches of grass can be rendered by computing necessary parameters for each pixel then applying the lighting model defined by the BTF. In [MLH02], McAllister uses the Lafortune lobes representation [LFTG97] for BRDFs which are the sum of a diffuse component and a set of specular lobes. The equation of the 4D function is:

$$f_r(\vec{\omega_i}, \vec{\omega_r}) = \rho_d + \sum_j \rho_{s,j} \cdot s_j(\vec{\omega_i}, \vec{\omega_r})$$

where $\vec{\omega_i}$ and $\vec{\omega_r}$ are the incidence and reflection directions, ρ_d the diffuse reflectance coefficient. Each specular lobe j has an albedo $\rho_{s,j}$ (reflectance factor) and a shape. The lobe shape is defined as follows:

$$s(\vec{\omega_i}, \vec{\omega_r}) = \left(\begin{bmatrix} \omega_{r,x} \\ \omega_{r,y} \\ \omega_{r,z} \end{bmatrix}^T \cdot \begin{bmatrix} C_x & 0 & 0 \\ 0 & C_y & 0 \\ 0 & 0 & C_z \end{bmatrix} \cdot \begin{bmatrix} \omega_{i,x} \\ \omega_{i,y} \\ \omega_{i,z} \end{bmatrix} \right)^n$$

$$s(\vec{\omega_i}, \vec{\omega_r}) = (C_x \omega_{i,x} \omega_{r,x} + C_y \omega_{i,y} \omega_{r,y} + C_z \omega_{i,z} \omega_{r,z})^n$$

The (C_x, C_y, C_z, n) coefficients define a specular lobe whose direction is the unnormalized vector $(C_x \omega_{i,x}, C_y \omega_{i,y}, C_z \omega_{i,z})$. If $C_x = -1$, $C_y = -1$ and $C_z = 1$, a standard Phong lobe is defined (see section 3.1.3 and figure 6*(a)*) with n as specular exponent. If $C_x = C_y$, the BRDF is isotropic,

Figure 10: Discrete BTF representation of a tree. The tree is rendered from different viewpoints (horizontal axis) with different light directions (vertical axis). Image from [MNP01].

otherwise it is anisotropic. This last property is very interesting for grass rendering since grass has anisotropic reflections in real life.

As explained in [MLH02], the Lafortune representation enables compact storage of a BTF with one texture map for the diffuse components and two maps for each specular lobe. The first map contains ρ_d RGB triples. The two other maps are used to store the (C_x, C_y, C_z, n) coefficients and the ρ_s RGB triples. In [McA04], McAllister explains how to implement the rendering step efficiently using fragment shaders that compute the local BRDF for each fragment. An advantage of this storage scheme is the per-texel BRDF representation.

With BRDF factorization methods such as some of those presented in [MMS+], only a palette of BRDFs can be created with their associated texture maps. Moreover, a rendering pass has to be performed for each BRDF and for each light. With the McAllister's representation [MLH02, McA04], the number of required rendering passes is independent of a palette size. Furthermore, each texel has its own BRDF representation offering high freedom.

The BTF gives a method to represent diffuse and specular components of a reflected flux from any point of a surface. When an ambient term is separately managed, night scenes can be rendered using a dark blue light coming from every directions. Likewise, the ambient term can represent the daylight coming from diffused sun rays inside the atmosphere. Diffuse and specular term would represent direct sun illumination.

In [MLH02, McA04], McAllister firstly gives a method to render a scene using discrete lights. A multi-pass algorithm is used with several passes per light source. He proposes a method that makes use of environment maps as a collection of infinite light sources, allowing for more accurate illumination since all incidence directions are taken into account. The radiance equation has to be used (section 3.1.3). Using environment maps for grass rendering allows for a better realism of the rendered scenes, particularly due to the contribution of light diffused inside the atmosphere. Beautiful sunset scenes may be obtained where the sky is blue on one side and pink on the other side.

Billboards

BTFs are also interesting for billboards. In [MNP01], Meyer et al. propose a method to render trees with a hierarchy of BTFs. A BTF is defined for each part of a tree: leaves, small branches, main branches and trunk. The hierarchy structure can be ignored for grass rendering to create a simpler representation where a patch of grass is located in the unique level. In figure 10, the texture is composed of a tree viewed from different points with various light directions. The tree can be replaced by a patch of grass. When rendering, the 3 nearest sampled view vectors are selected. The same action is applied to the light direction. A blending is performed with the 9 selected views taken from the pre-computed texture (as on figure 10). The blending coefficients are the barycentric coordinates relative to the view and light directions. Objects (trees, grass) are considered as directly seen from the camera, that is why the reflection vector used by the BTF is replaced by the view vector.

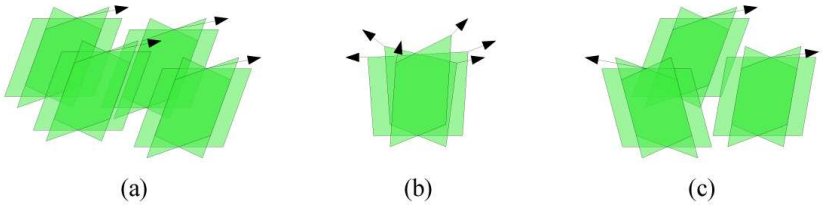

Figure 11: Three kinds of animation of grass objects from [Pel04]. *(a)* Per cluster of grass objects. *(b)* Per vertex. *(c)* Per grass object.

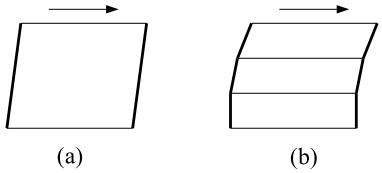

Figure 12: Two kinds of billboard deformation. *(a)* Quadrilateral billboard with shifted top. *(b)* Strip of polygons with progressive deformation.

3.2.4 Animation with IBR

As Image-Based Rendering uses static images, animation can not be performed in real-time applications. For animation purpose, new images have to be computed which could be very time-consuming when real-time rendering is targeted.

To simulate animation effect, one could deform geometry of the underlying terrain. In fact, terrain is not really deformed but the light reflection parameters are dynamically modified. Researches have to address this subject, which is also our objective.

To animate billboards, their associated quadrilaterals have to be bent. In [Pel04], Pelzer proposes to move the upper part of cross billboards (called *grass objects*) defined as on figure 8. The lower part should not be moved otherwise roots of grass blades would move and look unnatural. Pelzer proposes three kinds of deformation illustrated on figure 11. The first one consists in animating clusters of grass objects (figure 11*(a)*). Every top vertices of a cluster are shifted with a fixed translation vector. The drawback of this method is the apparent animation synchronization of grass blades inside a cluster. The second method consists of a per-vertex animation (figure 11*(b)*), easily performed with a vertex shader. Translation vectors are determined in different ways: pseudo-random generation, map of translation vectors, etc. However, this method produces more unnatural images than those of the previous one due to the distortion of the billboards polygons: length and thickness of grass blades are not preserved. The last method consists of per grass object animation (figure 11*(c)*) that shifts the top vertices of each independent grass object. This method offers many advantages: visual complexity due to local chaos, no distortions as for the second method, few draw calls because vertex shaders can be used with a map of translation vectors.

Perbet and Cani [PC01] employ a method similar to that by Pelzer [Pel04]. In addition, aligned layers of billboards are deformed. Not only the top of the billboards is animated (figure 12*(a)*) but also the middle levels using polygon strips (figure 12*(b)*). This method is efficient for distant grass viewed at walker's height. The same wind primitives [PC01], as pointed out at the end of section 3.1.4, define the amount of shift: slight breeze, gust of wind, whirlwind, blast of air. These primitives send parameters to the polygon strips to indicate how much they must be bent.

3.3 Volumetric textures

For many years, 2D textures has been often used to represent reflection parameters of surfaces. For flat surfaces, this representation is correct. However, the interaction of light with fur depends on the positions and directions of the fur's hairs. These latter can be considered as a *thick skin* over a surface with reflectance properties depending on the position in 3D space. This approach is well

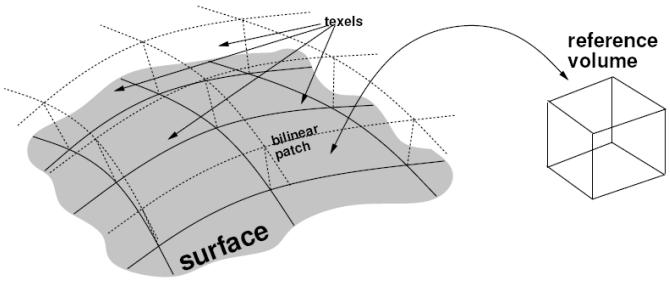

Figure 13: Reference volume repeated over a surface made of bilinear patches. Image from [Ney96]

suited for grass rendering: the surface to be rendered is the ground and the thick skin contains grass blades. Despite the fact that a high number of data are needed to represent grass accurately, volumetric rendering can be used for grass rendered at middle distance from the viewer since it usually offers pleasant results with few aliasing.

3.3.1 Volumetric rendering with raytracing

To render fur on objects like a teddy bear, Kajiya and Kay [KK89] introduced volumetric textures. They use a reference volume (that they call *texel*) which is repeated over an underlying surface made of bilinear patches (figure 13). The rendering method they use is raytracing. When a ray is cast and encounters an instance of the reference volume, it is projected inside the reference volume space and approximated by a straight line. Then, an optical model is used to determine color of the pixel to render. The ray is point sampled from front to back and the intensities of the traversed *voxels* (volume elements) are multiplied by a transparency cumulated along the ray. An area with *density* ρ (portion of space occupied by geometry) crossed on a length L has a *transparency* $e^{-\tau\rho L}$ where τ converts density to attenuation. The final intensity for a ray is $I = \sum_{near}^{far}(I_{loc}\prod_{near}^{current} e^{-\tau\rho L})$ (*optical equation*). The local illumination I_{loc} is the product of the incident light, the intrinsic reflectance and the *albedo* (reflected intensity depending on local density).

Neyret [Ney95b] introduces a multi-scale representation for more efficient rendering. When rendering, a coarse representation of data is used when the viewpoint is faraway, while a finer one is considered for a close viewpoint. Reference volume data are stored into an *octree*.

The advantage of volume rendering is that it handles very efficiently complex objects with many repetitive details. That is why it is well fitted to grass rendering. The multiscale filtered representation presented in [Ney95b] is subject to few aliasing effects. Volume representation offers full parallax: when the viewer is moving, objects are correctly rendered with no flatness as with billboards.

The main drawback of the raytracing method is the high rendering time it requires: several minutes to render a single image. Methods different from raytracing have to be used to meet the real-time constraint.

3.3.2 Real-time volumetric rendering

By discretizing the optical equation, classical hardware texturing capabilities can be employed. According to Ikits et al. [IKLH04], accumulated color and opacity (C and A) are computed as follows:

$$C = \sum_{i=1}^{n} C_i \prod_{j=1}^{i-1}(1 - A_j)$$

$$A = 1 - \prod_{j=1}^{n}(1 - A_j)$$

where C_i and A_i are the color and opacity of the i^{th} sample along the ray. A_i approximates the absorption. The opacity-weighted color C_i is an approximation of the emission and the absorption between samples i and $i+1$. As Ikits et al. propose in [IKLH04], these formulas can be iteratively

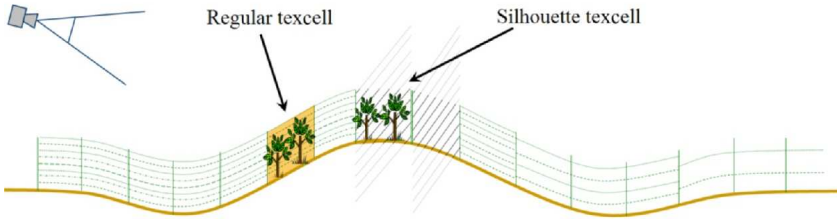

Figure 14: Slicing scheme for real-time rendering of *texcells*. Horizontal slicing is used except where the camera can view the slices (at grazing angle). Image from [DN04].

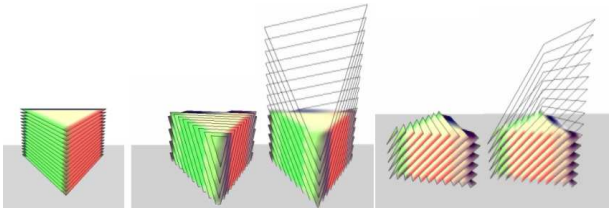

Figure 15: Types of texcells. Left: regular texcell. Middle: silhouette texcell facing the camera. Right: Side view of it. Image from [DN04].

evaluated for a rendering of the samples from back to front (*Over operator*) using the following equations:

$$\hat{C}_i = C_i + (1 - A_i)\hat{C}_{i+1}$$
$$\hat{A}_i = A_i + (1 - A_i)\hat{A}_{i+1}$$

For front to back rendering (*Under operator*):

$$\hat{C}_i = (1 - \hat{A}_{i-1})C_i + \hat{C}_{i-1}$$
$$\hat{A}_i = (1 - \hat{A}_{i-1})A_i + \hat{A}_{i-1}$$

$\hat{C}_i$ and $\hat{A}_i$ are the accumulated color and opacity from the front of the volume. These operations are efficiently performed using hardware texturing and blending, so enable real-time rendering. The previous method or slight variations of it are generally used for real-time rendering.

Decaudin and Neyret [DN04] use a similar method to render trees over hills. They pre-render trees inside a 3D texture and use it to cover the hills as on figure 14. The hills are subdivided in a bilinear patch, a 2D grid inside which each square contains the reference volume. This latter is called *texcell* (to avoid confusion with *texel* which is a term employed in other papers [KK89, Ney96] and which originally represents a *texture element*). Decaudin and Neyret [DN04] use two kinds of texcells: regular ones and silhouette ones, as shown on figure 15. The first ones use horizontal slicing, parallel to the ground, and are easy to manage. However, the spaces between slices can be seen at grazing angle. Thus, silhouette texcells are used whose slices are parallel to the view plane. Decaudin and Neyret [DN04] use vertex shaders to compute slices coordinates. Another method, described by Ikits et al. [IKLH04], only employs the CPU but is more precise: generated slices do not go beyond the limits of the volume texture. Decaudin and Neyret [DN04] propose using hardware culling to display only necessary parts of the volume (figure 15).

The previous method was originally designed to render trees over hills. It can be applied to grass using rendered grass blades inside the reference volume texture.

Publications about fur rendering give also interesting techniques. In [LPFH01], Lengyel et al. use series of semi-transparent textured *shells* to give the illusion of volumetric rendering (as on figure 16(middle)). These textures are created with slicing of the original geometry using clipping planes (figure 16(left)). Instead of using silhouette texcells as in [DN04], Lengyel et al. [LPFH01] use a *fin texture*, a texture containing a lateral view and used for the silhouette of objects. They need it because their fur rendering algorithm is dedicated to arbitrary surfaces and not to bilinear

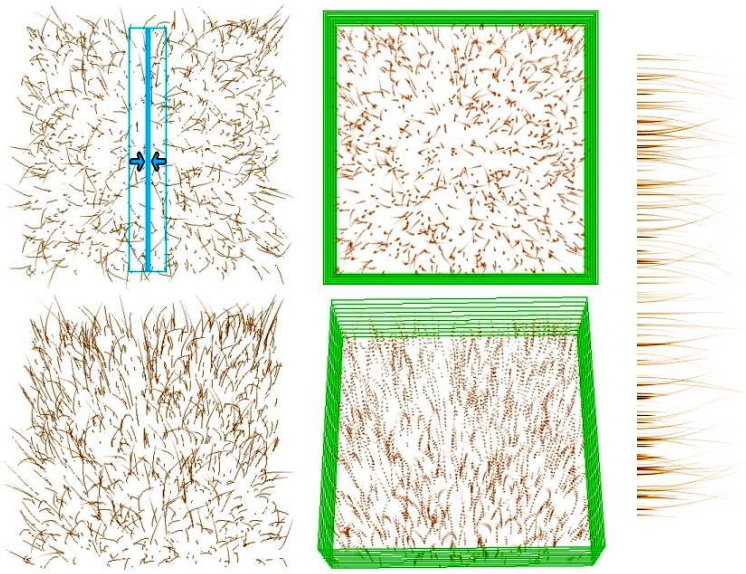

Figure 16: Fur shells. Left: geometry used to create the shells. Middle: stack of shells to give the illusion of volumetric rendering. Right: fin texture for the silhouette. Image from [LPFH01].

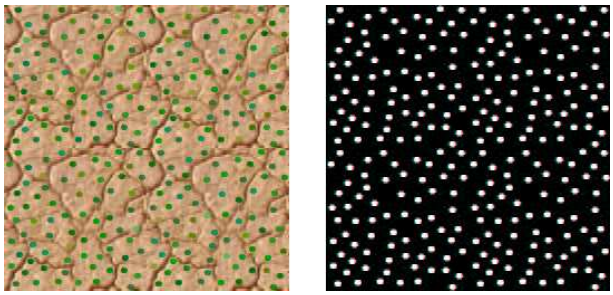

Figure 17: Bakay's shell texture [BLH02]. Left: RGB channel, texture of ground and cut grass blades. Right: alpha channel, representing grass blades positions; the gray level represents the height. Image from [BLH02].

patches. Grass rendering over bilinear patches can also use shells and fins textures to simulate volumetric texturing. Consequently, 3D texturing hardware is unused and then enables higher frame rates.

In [BLH02], Bakay et al. use also a stack of shells. However, their approach is less memory-intensive than Lengyel's one. They use a single texture to render ground and blades of grass with different lengths. An example of such a texture is given on figure 17. Quadrilaterals with the same texture are rendered several times with alpha test enabled and different alpha thresholds depending on the height. The texture is entirely displayed at ground level. For the upper shells, only the positions of the gray dots in alpha channel correspond to rendered texels. At the highest level, only maximal value of the alpha channel is used for rendering, corresponding to the highest blades of grass. This method has a bottleneck: fillrate. In spite of the relatively low number of rendered pixels due to the alpha test, shells rendering time is high because every quadrilaterals are completely rasterized. Moreover, this method does not manage lighting.

3.3.3 Data structures for volumetric textures

Data for volumetric textures can be stored in different ways. Each scheme has its advantages and drawbacks.

The first method is direct storage of the data inside a *3D array*. Each element of the array

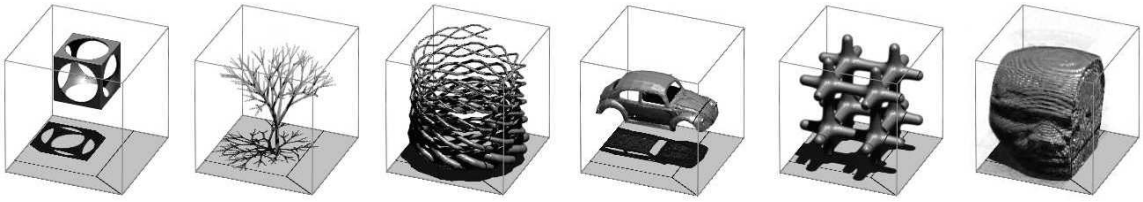

Figure 18: Volume conversion of: CSG (boolean operators applied to geometric primitives), L-system (procedural plants generation), particle system, mesh, implicit surface, tomographic image. Images from [Ney98].

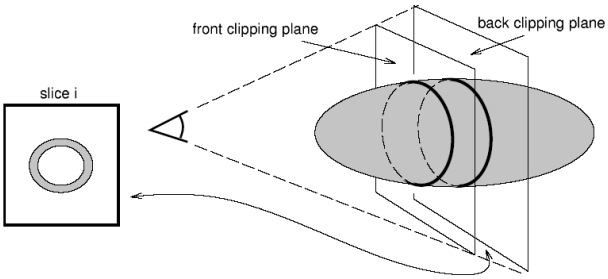

Figure 19: Slicing of geometry using a standard renderer with clipping planes. Image from [MN98].

contains opacity and reflectance parameters such as intrinsic color. It is the most complete representation but requires a large amount of memory and needs high bandwidth between the main processor and a 3D hardware.

A second representation is the *octree*, presented in section 3.3.1 and detailed in [Ney95b]. This hierarchical structure allows for lower memory consumption because some parts of the volume are not at maximal resolution. However, this representation is not suited to GPUs hence not for real-time applications.

Another representation is the *set of slices* presented in [MN98] by Meyer and Neyret and partially used in [DN04] to render forest landscapes (figure 14). A set of 2D textures stands for a volume. Yet empty spaces can be noticed between slices at grazing angle. An advantage is the possibility to render such slices with every 3D cards contrary to 3D textures which needs modern hardware.

We just presented different data structures for volumetric textures. These structures have to be filled. Neyret presents some methods in [Ney96, Ney98] (figure 18).

To convert a patch of grass to volumetric data, slicing with the standard hardware-accelerated renderer can be performed [MN98] (figure 19). For each slice of the destination 3D texture, the patch of grass is rendered between two clipping planes and the resulting pixels are transferred to the texture slice.

3.3.4 Lighting with volumetric textures

Volume rendering is a heavy task. That is why lighting inside volumes is often static in real-time applications. Precalculated lighting inside the reference volume is employed by Decaudin and Neyret in [DN04] to render trees on hills. The drawback is the impossibility to change the sun position.

Our goal is to render illuminated grass. Consequently, we will study some of the existing volume illumination methods.

As proposed by Neyret in [Ney95b], two kinds of information are stored inside the elements (*voxels*) of a reference volume: a density (similar to the amount of geometry inside the voxel) and a reflectance model. The latter can be separated into a generic reflectance model valid for the whole volume and a local orientation.

For Kajiya and Kay in [KK89], the reflectance model is represented by a thin cylinder, well

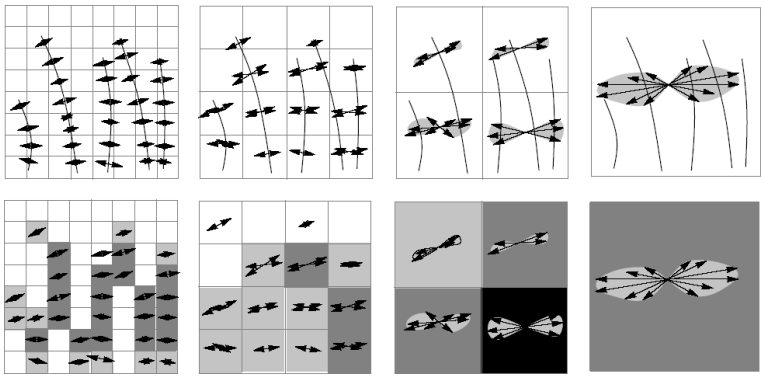

Figure 20: Blades of grass encoded at different resolutions (in an octree). The gray levels of the bottom figures represent the density. The arrows stand for the Normals Distribution Functions. The figures are represented in 2D for a clarity purpose. Image from [Ney98]

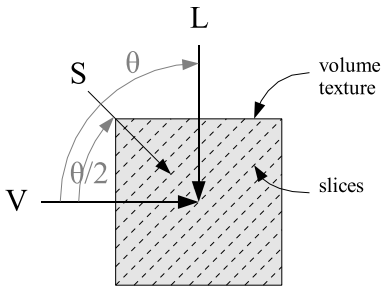

Figure 21: Slicing axis (S) halfway between the view (V) and light (L) directions. Image from [IKLH04]

suited for fur. The local orientations are equal to the fur's hairs directions.

Neyret [Ney95b] uses ellipsoids to represent a more general reflectance model. With them, he can model different shapes, from sphere to approximated cylinders. Furthermore, the ellipsoids are suited to a multiscale structure because it allows for filtering: several ellipsoids from a level can be approximated by a single ellipsoid of the upper level.

The two last reflectance models are in fact special cases of general functions known as *Normals Distribution Functions (NDFs)* [Ney98]. They represent the distribution of normals to the surfaces inside a voxel. The figure 20 illustrates NDFs management inside an octree [Ney95b, Ney98].

The previous lighting techniques are originally dedicated to raytracing rendering. Variations of these algorithms can be used for real-time applications.

By using slices of the reference volume (section 3.3.2), it is possible to compute illumination inside a volume with scattering in real-time. A pixel buffer can be employed to accumulate the amount of light attenuated from the light's point of view [IKLH04, KPH$^+$03]. The slicing axis is set halfway between the view and light directions (figure 21). For each rendered slice, two passes are performed. The first one consists in rendering the slice from the eye's point of view, modulating the color by the opacity stored inside the pixel buffer. The second pass accumulates the opacity to the pixel buffer by rendering the old slices and the new one from the light's point of view. This method is efficient and is suited to grass rendering using hardware 3D texturing.

3.3.5 Animation with volumetric textures

As defined at the beginning of section 3.3, space-deformations of a reference volume (texcell) mapped onto a surface create a thick skin of complex objects. Neyret introduces simple methods in [Ney95a] to animate these texcells: deformation of the underlying surface (top of figure 22), animation of height vectors allowing for deformation of the whole volume space (middle of figure 22), successive steps of a simple motion using a set of reference volumes (bottom of figure 22).

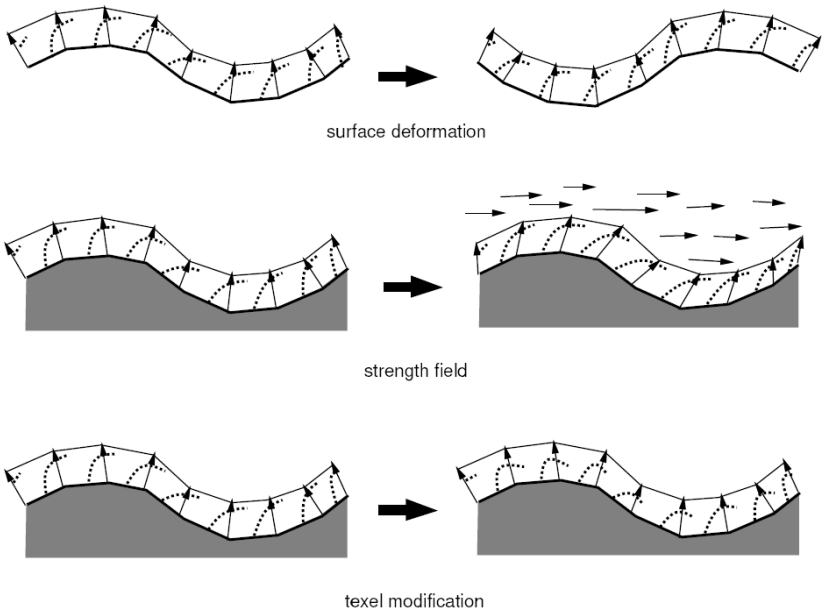

Figure 22: Three ways of animating texcells. Image from [Ney95a].

The first method is not applicable to grass since the underlying terrain does not move. The second method is easy to manage: only a set of vectors has to be updated each frame. The vectors can be jittered by force fields. These fields can be provided by physical simulation or ad hoc functions. Usually, ad hoc functions inspired by real grass behavior are defined, they are faster to compute than pure physical simulation. The Perbet and Cani wind primitives [PC01] are applicable to height vectors jittering (as for image-based rendering).

The last method needs a large amount of memory. It can be interesting for cyclic movements of grass blades with low amplitude. However, movement is always the same and bad transitions can occur between texcells. To avoid these latter, animation has to be synchronized between texcells, creating large surfaces of grass with the same movement which looks unnatural.

These three methods introduced by Neyret [Ney95a] were dedicated to raytracing rendering. In [MN98], Meyer and Neyret use slicing to render grass at interactive rates and apply these animation techniques, particularly the second one.

4 Our proposed solution

4.1 Strategy

Grass rendering has been studied since many years. Numerous techniques have been designed to render grass at different distances from the viewer. The real-time techniques have already been collected inside the implementation of Perbet and Cani [PC01] with dynamic management of the levels of detail. However, animation was their main purpose. Our goal is to create a grass renderer with the integration of illumination models to create more natural looking images.

A second goal is to create a modular grass rendering library. We want to be able to add rendering algorithms, illumination models, etc. without modifying a large part of the code already written. Moreover, it is easier for other users to implement future research results into the library.

We firstly give the algorithms we plan to use to render grass. Then we briefly present the software architecture we designed for the moment.

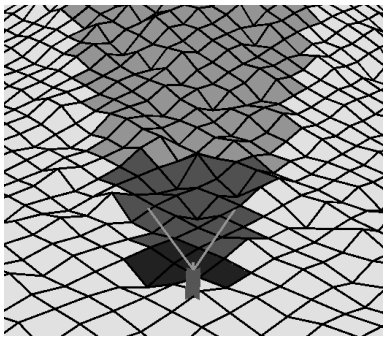

Figure 23: The terrain is tiled into patches of grass. The walker has a view frustum that determines which tiles must be displayed. Then the three shades of gray represent the 3 main rendering methods: dark gray for geometry, medium gray for volumetric textures and light gray for 2D textures. Image from [PC01].

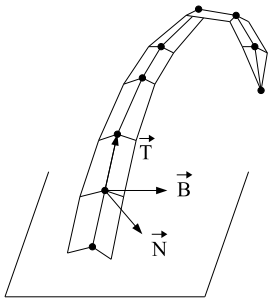

Figure 24: Grass blade generated by a particle system and displayed with quadrilaterals and triangles.

4.2 Proposed algorithms

4.2.1 Levels of detail (LOD) management

When rendering very complex scenes, computing resources can be quickly saturated. Levels of detail allow for drop in rendering quality to improve rendering speed of far objects.

In our approach, we define two depths of levels of detail:

- *coarse* levels: we think of using different kinds of rendering methods: geometry for the closest grass blades to the camera, volumetric rendering when distance increases and 2D textures for the farthest grass. An example of LOD management is shown on figure 23.
- *fine* levels: to further improve rendering speed, levels of detail can be used within each method. We present them in the following subsections.

The level of detail to use for a rendered patch of grass have to be dynamically determined. The easiest criterion is the distance from the camera. However this approach is efficient only if the camera height is almost constant (from walker's height for instance). If the camera goes to the sky (for a fly-through), the criterion has to be modified.

4.2.2 First level: geometry

In section 3.1.2, we presented particle systems. The first method we plan to use is illustrated on figure 24. The trajectory of an individual particle is used to generate the shape of a grass blade defined by surface primitives. At each simulation step, a local basis $(\vec{T}, \vec{N}, \vec{B})$ is determined and used to compute the coordinates of two quadrilaterals. At the end of the grass blade, two triangles are used to create a sharp end. The generated geometry can be visually more detailed if classical texturing is used. As presented in section 3.1.3, we can use subsurface scattering to greatly improve realism. Hypothesis can be formulated about grass blades to simplify the depth map technique

presented in [Gre04] to improve frame rate. Indeed, grass blades have almost constant thickness and are thin.

Rendering with geometry needs a high computing power when thousands of grass blades have to be displayed at the same time. That is why this method is dedicated to grass which is very close to the camera.

For geometric blades located a bit farther from the camera, we can reduce the precision of the geometry by decreasing the number of steps of the particle system simulation when pre-processing. The curvature of grass blades will be coarser but with the sampling density of the screen, it should be unnoticeable.

Textures applied over the grass blades can employ classic mipmapping techniques to reduce aliasing and improve the frame rate. Texturing may be disabled when the distance is too long and replaced by colored vertices interpolation.

As in [RB85], anti-aliased lines can be drawn to render the shape of grass blades. This method should be tested experimentally to prove its efficiency.

4.2.3 Second level: volumetric textures or billboards

In sections 3.2 and 3.3, we presented billboards and volumetric textures. These two rendering techniques are similar if billboards are considered as slices inside a volume.

We plan to use a mix of these methods for grass at middle distance from the camera. Real-time approach for volumetric rendering of Decaudin and Neyret [DN04] using regular and silhouette texcells is interesting for the full parallax property. To introduce more chaos, vertical billboards as those used by Perbet and Cani [PC01] can be added.

To manage grass density, threshold values can be associated with the billboards pixels and the voxels. When rendering, if the user-defined grass density at a point of terrain is greater than the threshold of a pixel, the latter can be rendered.

4.2.4 Third level: 2D textures

2D textures rendered onto the terrain faces are used for grass standing in a long distance from the camera. The BTF technique [MMS+, Dis98, McA04] will be used to simulate illumination due to lighting conditions (as explained in section 3.2.3). Per-pixel varying illumination can simulate animation due to wind.

The set of textures needed by the BTF method will be created using the geometry renderer with varying lighting conditions and view directions. A set of samples is then obtained for each local BRDF, model fitting has to be performed to approximate these sampled BRDFs by Lafortune lobes [LFTG97]. This representation is non-linear so non-linear model fitting is needed. This fitting is introduced in [MMS+].

4.2.5 Transitions between levels of detail

Performing a smooth transition between the levels of detail is a difficult task. Generally, popping artifacts appear when the viewer is moving.

The first step to reduce these artifacts is the use of the same patch of grass for each representation: geometry, billboards, volume, 2D textures. That is why the three last representations require procedural generation using geometry rendering.

A transition scheme is introduced by Perbet and Cani in [PC01]. Instead of using cross dissolve between geometry and billboards which gives poor results, they use a morphing technique. Their method is applicable to grass with motion. They define another method for transitions between billboards and 2D textures: instead of directly applying 2D textures onto the terrain, they apply them onto an offset of the terrain, located between the top and the bottom of the grass blades.

4.3 Software architecture

We want to design a modular grass rendering library. Thus, components have to be designed with their own purpose and relationships with other components have to be brought out. The software architecture we propose is illustrated on figure 25.

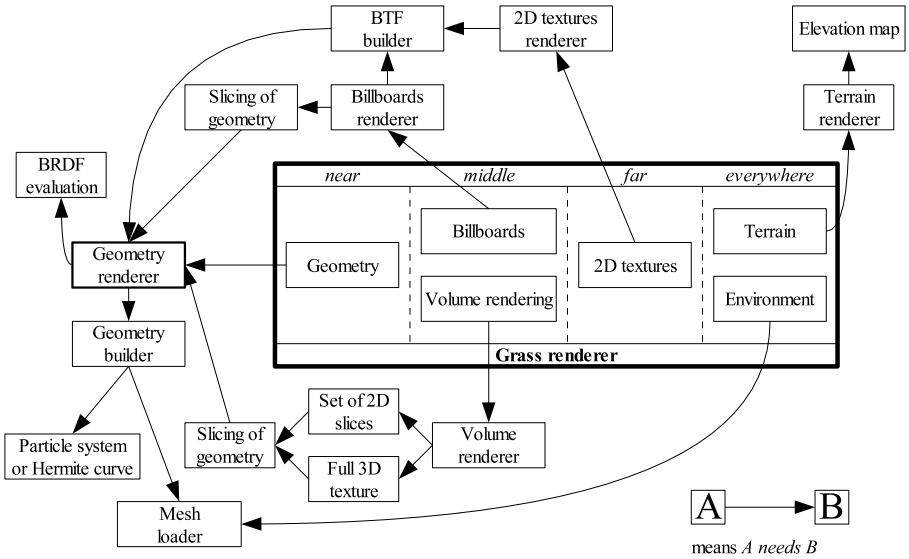

Figure 25: Planned software architecture of our grass renderer.

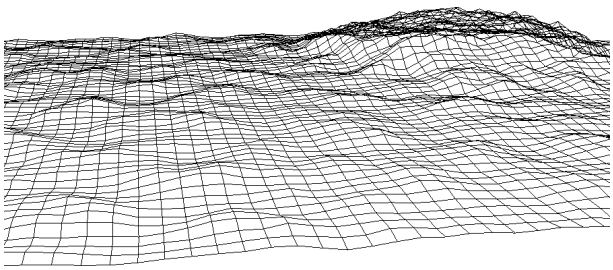

Figure 26: Terrain modeled with a regular grid and an elevation map. Image from [DHL+98]

Firstly, we need a set of renderers, one for each rendering method: geometry, billboards, volume, 2D textures. Then, data are provided by loaders (meshes, images) and builders (geometry, BTFs). On figure 25, the geometry renderer frame has a bold border because it is the most important component: every rendering methods need it, either for direct rendering or for data building. Consequently, it is one of the first components we plan to develop.

Some additional components [DHL+98] are required before the rendering stage: editors (terrain, grass density, lights, etc.) and managers (textures, environment actors). At the beginning, these components can be replaced by scripts for example. Editors and managers with a graphical user interface (GUI) will be designed later.

4.4 Terrain modeling and rendering

In real life, grass is always growing from roots inside ground. To produce realistic images with grass, a terrain has to be rendered. *Elevation maps* allow to model the vertices height of a tiled terrain [PC01, Pel04] (as on figure 26). Each value of the elevation map stands for the height of a terrain vertex. Inside each tile, a patch of grass is defined as on figure 23. We can use this kind of technique to easily manage the levels of detail as presented in section 4.2.1.

The terrain can be modeled using a regular grid of quadrilaterals or by an adaptive method like *geomipmapping* [dB00]: the terrain faces topology changes depending on a criterion based on the distance from the camera and the projection onto screen (to reduce popping artifacts). The Boer's method is well suited for us because it is designed to minimize the CPU's load.

At the beginning we plan to use the simplest method, elevation maps with a regular grid, because terrain rendering is not the bottleneck of a grass renderer.

5 Possible enhancements

We present here some ideas that can be exploited to enhance the rendering quality. We plan to implement them if the main algorithms are functional.

- *Glittering*: when grass is wet due to rain or dew, glittering can be observed when the viewer moves. The BTFs used to render grass at middle and far distances uses interpolation of samples rendered with geometry, the displayable frequencies are then low. The glittering is a high frequency phenomenon. It can be implemented used an additional rendering pass over the rendered grass.
- *Shadows from environement*: in real life, grass is never alone, there are always other actors in the environment (trees, bushes, rocks, animals, paths, statues, bridges, etc.) which cast shadows. These latter has to be projected onto grass to give a natural looking result. Grass also casts shadows onto the environment.
- *Ground texture variations*: in real-life, ground aspect is different at each location. Several textures have to be used with random or user-defined locations.
- *Management of multiple light sources*: by day, it is not necessary because sunlight is very bright and so can be considered as the only light source. By night, it can be interesting to manage lighting due to multiple street lamps. In this case, multi-pass rendering may be necessary for image-based and volumetric rendering.
- *Atmosphere blue shift*: in real life, a blue shift appears when viewing very far objects. This is due to the absorption of the light spectrum red part by the atmosphere. It can be easily simulated with a blue fog using hardware fog capabilities. Realism can be further improved by using advanced techniques such as volumetric fog.

6 Conclusion

Rendering natural environments is a difficult problem in computer graphics, because of its high complexity. In this report, we have presented techniques that make some approximations to achieve real-time grass rendering. We are planning to implement some of them.

However, lighting simulation has often been ignored because it is highly demanding in terms of computing power. To compute light reflections, we have introduced some reflectance models of general usage in computer graphics. To get naturally looking images of grass, our goal is to integrate these reflectance models into some of existing grass rendering methods.

Moreover, we intend to apply animation techniques to grass. Making an efficient use of these techniques, when the purpose is interactivity, is a very difficult challenge.

Using precise lighting models and performing realistic animation within the same real-time application is not tractable even with the existing high performance graphics cards. Our solution will have to make a compromise between the quality of lighting and that of animation.

Grass rendering seems to be a research subject of little interest. In fact, numerous video games companies are interested in it. Nowadays, an important competition exist between these companies to create the most beautiful games. Studying grass rendering will allow for more natural looking images since more and more games take place in outdoor environments. Other companies such as those creating flight simulators are also interested in grass rendering, particularly in the management of levels of detail.

References

[AMH02a] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering - Second Edition*, chapter 12.1.3, pages 492–494. A K Peters, 2002.

[AMH02b] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering - Second Edition*, chapter 6.3, pages 194–206. A K Peters, 2002.

[BLH02] Brook Bakay, Paul Lalonde, and Wolfgang Heidrich. Real time animated grass. In *Eurographics 2002*, 2002.

[dB00] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping, October 2000.

[DCSD] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. Interactive visualization of complex plant ecosystems.

[DHL+98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *ACM Computer Graphics Proceedings*, Annual Conference Series 1998, pages 275–286, 1998.

[Dis98] Jean-Michel Dischler. Efficiently rendering macro geometric surface structures with bi-directional texture functions. In *Eurographics Workshop on Rendering 98*, pages 169–180, 1998.

[DN04] Philippe Decaudin and Fabrice Neyret. Rendering forest scenes in real-time. In A. Keller H. W. Jensen, editor, *Eurographics Symposium on Rendering*, 2004.

[GPR+] Sylvain Guerraz, Frank Perbet, David Raulo, François Faure, and Marie-Paule Cani. A procedural approach to animate interactive natural sceneries.

[Gre04] Simon Green. *GPU Gems*, chapter 16 - Real-Time Approximations to Subsurface Scattering, pages 263–278. Addison-Wesley, March 2004.

[IKLH04] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. *GPU Gems*, chapter 39 - Volume Rendering Techniques, pages 667–692. Addison-Wesley, March 2004.

[KK89] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In *Computer Graphics (SIGGRAPH 89 Proceedings)*, volume 23(3), pages 271–280, July 1989.

[KPH+03] J. Kniss, S. Premo, C. Hansen, P. Shirley, and A. McPherson. A model for volume lighting and modeling. In *IEEE Transactions on Visualization and Computer Graphics*, volume 9(2), pages 150–162, 2003.

[LFTG97] E. P. F. Lafortune, S.-C. Foo, K. E. Torrance, and D. P. Greenberg. Non-linear approximation of reflectance functions. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 117–126. ACM Press/Addison-Wesley Publishing Co., 1997.

[LPFH01] Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. In *ACM Symposium on Interative 3D Graphics*, pages 227–232, March 2001.

[McA04] David McAllister. *GPU Gems*, chapter 18 - Spatial BRDFs, pages 293–306. Addison-Wesley, March 2004.

[MDK99] Nelson Max, Oliver Deussen, and Brett Keating. Hierarchical image-based rendering using texture mapping hardware. In *Eurographics Workshop on Rendering 99*, pages 57–62, 1999.

[MLH02] David K. McAllister, Anselmo Lastra, and Wolfgang Heidrich. Efficient rendering of spatial bi-directional reflectance distribution functions. In *Graphics Hardware 2002, Eurographics / SIGGRAPH Workshop Proceedings*, 2002.

[MMS+] G. Müller, J. Meseth, M. Sattler, R. Sarlette, and R. Klein. Acquisition, synthesis and rendering of bidirectional texture functions.

[MN98] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop 1998*, pages 157–168, July 1998.

[MNP01] Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. Interactive rendering of trees with shading and shadows. In *Eurographics Workshop on Rendering*, pages 183–196, July 2001.

[Ney95a] Fabrice Neyret. Animated texels. In *Eurographics Workshop on Animation and Simulation 95*, pages 97–103, September 1995.

[Ney95b] Fabrice Neyret. A general and multiscale model for volumetric textures. In *Proceedings of Graphics Interface 95*, pages 83–91, May 1995.

[Ney96] Fabrice Neyret. Synthesizing verdant landscapes using volumetric textures. In X. Pueyo and P. Schroöder, editors, *Rendering Techniques 96*, pages 215–224 and 291. Springer-Verlag, 1996.

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

[PC01] Frank Perbet and Marie-Paule Cani. Animating prairies in real-time. In S.N. Spencer, editor, *Proceedings of the Conference on the 2001 Symposium on interactive 3D Graphics*. Eurographics, ACM Press, 2001.

[Pel04] Kurt Pelzer. *GPU Gems*, chapter 7 - Rendering Countless Blades of Waving Grass, pages 107–121. Addison-Wesley, March 2004.

[RB85] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH 85 Proceedings)*, volume 19(3), pages 313–322, July 1985.

[Ree83] William T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects, July 1983.