PUBLICATION
INTERNE
N^{o} 1809

RENDERING GRASS IN REAL-TIME WITH DYNAMIC LIGHT SOURCES AND SHADOWS

KÉVIN BOULANGER AND SUMANTA PATTANAIK AND KADI BOUATOUCH

ISSN 1166-8687

IRISA

Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Rendering Grass in Real-Time with Dynamic Light Sources and Shadows

Kévin Boulanger [*] and Sumanta Pattanaik [**] and Kadi Bouatouch [***]

Systèmes cognitifs
Projet Siames

Publication interne n1809 — Juillet 2006 — 33 pages

Abstract: Since grass is very abundant on the Earth's surface, it is an important element of natural 3D scenes. Real-time realistic rendering of grass has always been difficult due to the huge number of grass blades. Overcoming this geometric complexity usually requires many coarse approximations to provide interactive frame rates. However, the performance comes at the cost of poor lighting quality and lack of detail of the grass. In this report, we describe a grass rendering technique that allows better lighting and parallax effect while maintaining real-time performance. We use a novel combination of geometry and lit volume slices, composed of Bidirectional Texture Functions (BTFs). BTFs, generated using a fast pre-computation step, provide an accurate, per pixel lighting of the grass. Our implementation allows the rendering of a football field, covered by approximately 627 million virtual grass blades, with dynamic lighting, shadows and anti-aliasing in real-time. The creation of arbitrary shaped patches of grass is made possible using our density management.

Key-words: grass rendering, real-time, lighting, shadows, levels of detail, density, volume rendering, BTF, anti-aliasing

(Résumé : tsvp)

[*] kboulang@irisa.fr
[**] sumant@cs.ucf.edu
[***] kadi@irisa.fr

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Rendu d’Herbe en Temps Réel avec Eclairage dynamique et Ombres

Résumé : Puisque l’herbe couvre une grande partie de la terre, il est important de l’inclure dans les scènes naturelles 3D. Le rendu en temps réel d’herbe a toujours été une tâche difficile à cause du nombre très élevé de brins d’herbe à gérer. Pour contourner cette difficulté, on a souvent recours à de fortes approximations lorsque l’objectif est l’interactivité. Malheureusement, ces performances sont obtenues au détriment de la qualité de l’éclairage et la finesse des détails de l’herbe. Dans ce rapport, nous proposons une méthode de rendu d’herbe offrant un meilleur réalisme de l’éclairage et un effet de parallaxe tout en assurant une performance de temps réel. Nous utilisons une nouvelle méthode combinant géométrie et tranches de volumes, ces dernières étant représentées par des BTFs (Bidirectional Texture Functions) générées dans une phase de prétraitement. Cette méthode assure un calcul précis par pixel de l’éclairage. Notre mise en oeuvre permet le rendu d’un terrain de football couvert par plus de 627 millions de brins d’herbe, un éclairage dynamique, un calcul d’ombre et un anticrênelage en temps réel. La créatioon de surfaces d’herbe de forme quelconque est rendue possible par notre méthode de gestion de density d’herbe.

Mots clés : rendu d’herbe, temps réel, éclairage, ombres, niveaux de détail, densité, rendu volumique, BTF, anticrênelage

Contents

1 Introduction

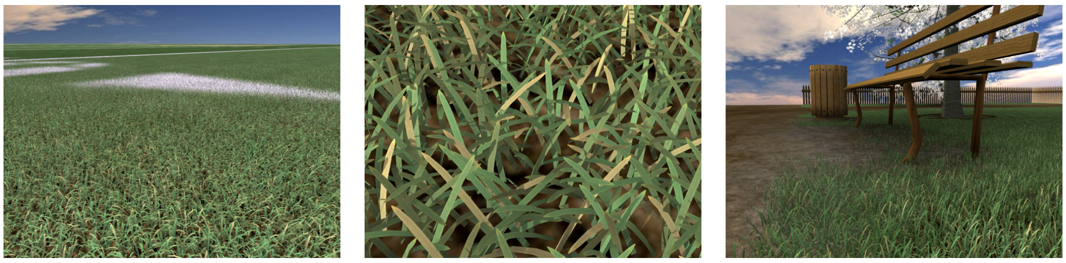
Figure 1: From left to right: football field with 627 million grass blades, close view where shadows are visible, park scene with user-defined grass density.

Grass is the plant family that occupies the greatest area of the world's land surface. It can be found in meadows, prairies, forests, mountains, savanna, football stadiums, etc. It is then crucial to account for it when rendering natural 3D scenes.

A surface of grass is composed of a large number of *grass blades*, too large to be fully stored in memory and rendered directly. Our goal is to render surfaces of grass with the highest obtainable fidelity, at real-time frame rates. Overcoming the complexity of grass rendering has been a challenging problem for many years. Previous approaches either render grass in real-time [1, 2, 3, 4, 5] but with coarse approximations, or render grass in high quality but offline [6]. We propose an approach that allows real-time rendering of large surfaces of grass with dynamic lighting, dynamic shadows and anti-aliasing. Rendering high quality grass with accurate lighting is still not possible even with high performance graphics cards. Our levels of detail approach provides a good compromise between lighting quality and rendering speed, and allows changing the balance for a better quality or a better speed. Arbitrary shaped surfaces of grass can be easily created using our management of grass density.

This report is structured as follows. First, we present some background notions we use in our approach, with their pros and cons. Next, we present our solution step by step. Then, we introduce some implementation issues and how we solved them. Finally, we present some rendering results followed by a conclusion and proposals for future work.

2 Previous work

Geometry-based rendering methods allow precise representations of objects, with textures and lighting. However, the processing power needed to render many million grass blades in real-time is not yet available. So far, only offline processing has been used to render geometric grass, in particular for movies.

Image-Based Rendering (*IBR*) is often used when geometry is too complex to be efficiently rendered. Billboards are the simplest of the IBR approaches. They are triangles or quadrilaterals covered by a semi-transparent 2D texture. They allow efficient rendering of complex natural objects such as trees. Their rendering is more efficient than classical geometry since a single primitive can be used in place of a large amount of geometry. Several kinds of billboards have been designed: single quadrilateral rotated to be always aligned with the camera [7], fixed aligned layers of quadrilaterals [2], fixed crossed quadrilaterals [4]. The main drawback of these methods is the lack of parallax effect. Thus using them makes difficult the creation of view-dependent realistic rendering.

Rendering billboards using simple semi-transparent textures does not allow complex lighting. Parameters such as reflection properties are missing. In our approach, we make use of a modified version of *Bidirectional Texture Functions*, also called *BTFs* [8]. They represent reflectance properties of a surface point depending on its position on the surface, the view direction and the incident light direction. They are an extension of *BRDFs* (*Bidirectional Reflectance Distribution Functions*) [9], which are constant for every point of a surface. A spatially varying BRDF is called *SBRDF* [10]. If the BRDF per point of the surface is multiplied by a visibility function, we obtain an *Apparent BRDF* (*ABRDF*). A BTF is composed of such ABRDFs for every point of a surface, allowing the management of self-shadowing and self-occlusions. If an *alpha* channel is added to the BTF data, this reflectance function can be used for billboards, rather than simple semi-transparent textures. A different way to represent a BTF, also adapted to billboards, is the use of a set of images per view and light direction [11].

A third method of rendering complex objects with repetitive details is volume rendering [12, 13]. A 3D reference volume, usually a box, contains one or more instances of the object that has to be rendered. This volume is tiled over an underlying surface. Volume representation offers full parallax: when the viewer is moving, objects are correctly rendered with no flatness impression as with billboards. Volume rendering has already been used to render grass [13] but not in real-time. Generally, raytracing is used to display these volumes. Other volume rendering methods have been used to meet the real-time constraint [14, 15, 16, 17], in particular using 2D textured slices. Bakay et al. [3] define a simpler approach, based on slicing, using a single texture to render the ground and the blades of grass with different lengths. The texture contains the image of the ground and green dots. A stack of quadrilaterals with the same texture is rendered with alpha test enabled. Different alpha thresholds create different blade lengths. However, all grass blades look similar and this method does not manage lighting.

The method presented in this paper uses a combination of geometry-based and volume-based approaches, the volume data being defined using BTFs. We outline the method in the following section.

3 Our grass rendering method

The main goal of our work is to design and develop a GPU-based grass rendering system that integrates dynamic illumination and good parallax effect in real-time applications containing very large surfaces of grass. We combine geometry and volume rendering using a levels of detail scheme (*LOD*) to achieve this goal. We start by presenting our global levels of detail scheme. Then, we give details about the rendering method for each level. Next, we present the density management allowing the creation of non-uniform grass distributions and the management of smooth transitions between levels of detail. Finally, we describe our shadowing algorithm.

3.1 Levels of detail

We want to render large terrains covered with grass, football fields for example. Direct rendering of geometric blades of grass with lighting is impossible in real-time (about 4 minutes per frame for a soccer field made of 250 million grass blades), so we need to make use of levels of detail. We use the distance from the camera as a criterion to switch between levels (Figure 2). If the grass is close to the camera, the best rendering quality is used, performed using lit and shadowed geometry. When grass is farther, a large number of grass blades covering a few pixels have to be rendered. We use an approach faster than geometry: volume rendering using semi-transparent axis-aligned slices.

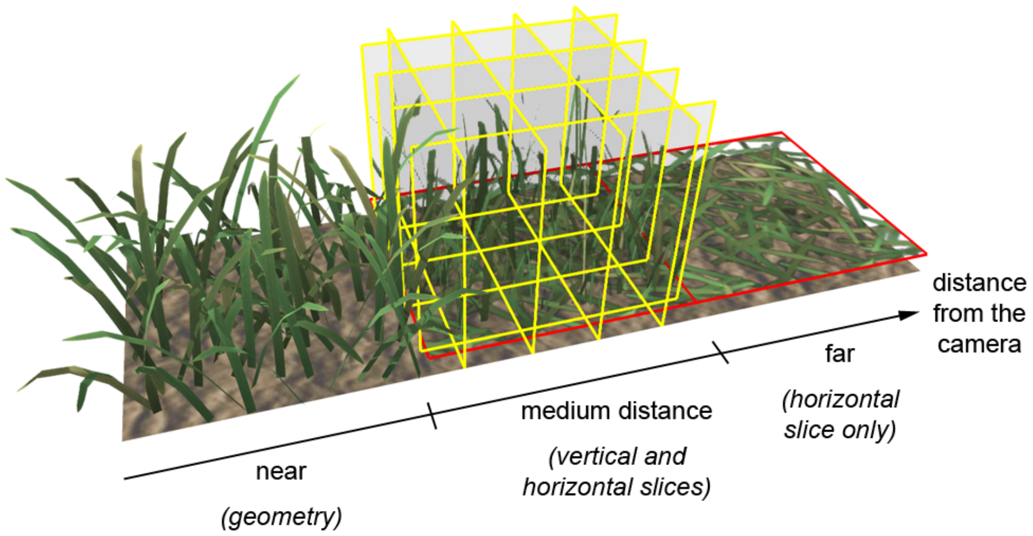

Figure 2: The three levels of detail, chosen depending on the distance from the camera. For nearby grass, simple geometry is used. At moderate distance, horizontal and vertical semi-transparent slices are rendered. For faraway grass, only the horizontal slice is kept.

However, we cannot define all parameters of every blade of grass of a terrain because the required memory amount is excessive. We need to define a smaller primitive and use it several times: we render several instances of a *grass patch* over the ground surface [12, 13], laying on the cells of an uniform grid. We define this grass patch two different ways: as a set of geometric grass blades distributed inside a rectangle, and as a set of axis-aligned slices using semi-transparent textures (Figure 2). The latter approach offers a good parallax effect: seen from any direction, the grass patches do not look flat. For very far grass, the number of slices to be rendered is excessive. At this distance, a surface of grass looks flat, thus only the slices parallel to the ground are kept.

A difficulty of any LOD scheme lies in the seamless transition management. We describe in Section 3.4 the way we perform smooth transitions between levels.

There can be significant visual differences between levels of detail if the data representing these levels are different. For instance, we cannot use an external high-quality raytracer to generate the volume slices data, otherwise variations of color would be visible at the transition between geometry-based and volume-based grass. In our approach, the generation of data for the volume slices is done by rendering a patch of geometry-based grass, detailed in Section 3.3.

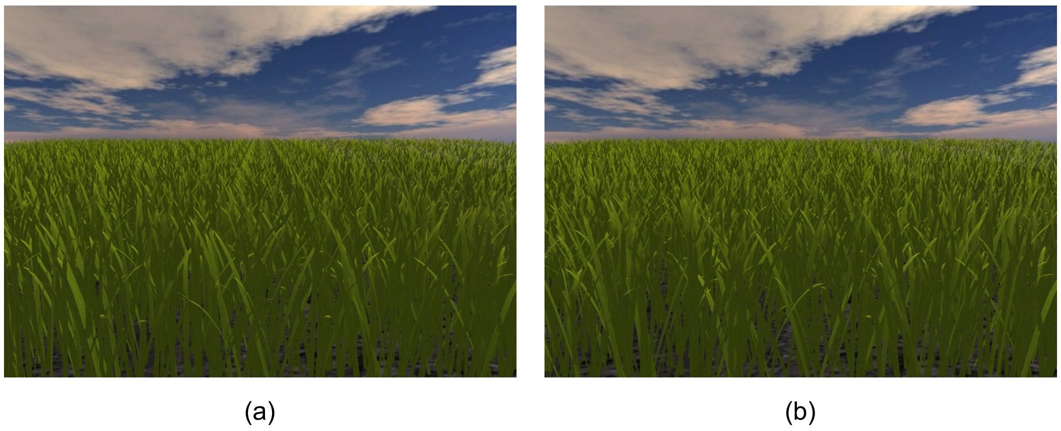

Figure 3: Result of aperiodic tiling. *(a)* Repetition of a single grass patch without symmetries (periodic tiling), repetitive patterns can be observed at the center of the image. *(b)* Random symmetries of the grass patch instances to achieve aperiodic tiling. No repetition pattern can be observed.

Repeating the same patch of grass many times over a whole terrain generates a distracting visual pattern. We introduce a simple *aperiodic tiling* scheme that consists in using four different versions of the unique grass patch. Then each version is used randomly for each terrain grid cell (Figure 3). We define the four patches as mirrored versions of the base patch, so data are present only once in memory, the symmetry operation being done at run-time in the GPU. Such random symmetries break the strong visual pattern enough. No problems are visible at the patch borders: the roots of grass blades defined by geometry are inside the patch bounds but the tips can go outside, and interleave with the blades of the neighbor patches. For faraway grass, the high visual complexity hides the transitions. By following this simple aperiodic tiling scheme, we get a reasonably good result and the rendering speed is almost not affected.

If the distribution of the grass blades is not uniform inside the grass patch, the clumps of grass that are more dense appear in a regular fashion over the terrain, even using random patch symmetries. The distribution of grass blades inside a patch has to be as uniform as possible while keeping the random distribution. Distributing grass blade roots using a random numbers generator for their coordinates does not give good results: the number of roots should theoretically be infinite to achieve the uniformity. To improve this uniformity, we use *stratified sampling*: the grass patch is subdivided into a fine uniform grid and a grass blade is placed at a random location inside each cell.

3.2 Geometry-based rendering

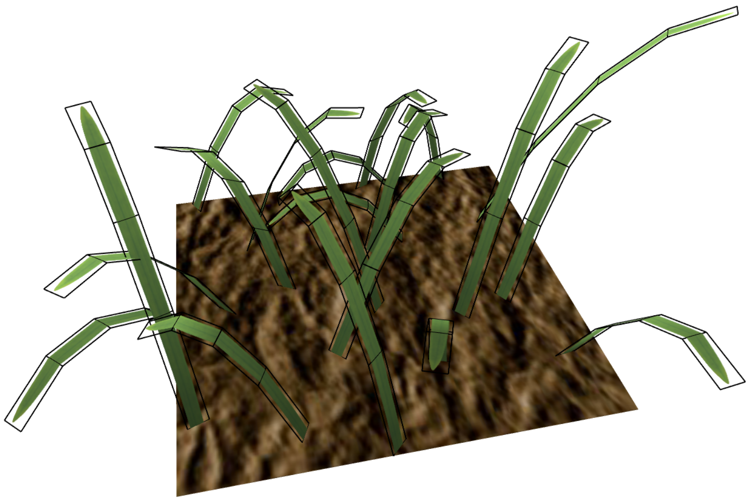

Figure 4: Grass blades defined with textured semi-transparent quadrilateral strips.

Geometrically modeled grass blades are used for rendering close to the camera and for the generation of volume slices data. We model a grass blade by a *quadrilateral strip* as shown in Figure 4. Real grass blades are very thin: we approximate them with two-sided quadrilaterals of zero thickness. The trajectory of a particle system [18, 1] defines the shape of the strip: a particle is launched from the root of the blade, almost vertically, with influence of the gravity. The particle position is evaluated several times, giving the coordinates of the blade reference points from which we determine the vertex coordinates. The *alpha* channel (Figure 5*(c)*) of the texture covering the quadrilateral strips (Figure 5*(b)*) gives the correct shape of a grass blade. In Section 4.2, we detail our method to perform order-independent rendering of the semi-transparent quadrilaterals used for the grass blades.

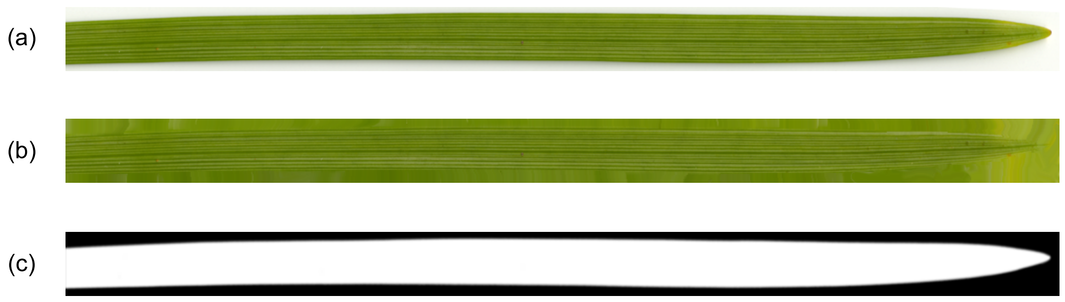

Figure 5: Texture used for a grass blade. *(a)* Original scanned blade. *(b)* Color channel of the texture modified to remove the white border. *(c)* Alpha channel of the texture defining the shape of the blade.

The Lambert reflection model is used for the grass blade surfaces, representing reflectance of diffuse only surfaces. An ambient component is added to partially simulate lighting from the environment and inter-reflections. For each blade, two-sided lighting is enabled: the face to be rendered (front or back) is selected depending on the normal vector projections in camera space. The color of each blade is slightly modified to simulate different ages and levels of degradation. To simulate ambient occlusion, we set the color of the blades darker close to the ground [1] because the amount of occlusions due to the neighbor blades is higher than for the blade tips. The ambient occlusion coefficient per vertex is calculated as a linear function of the height of the vertex from the ground.

3.3 Volume rendering

We use volume rendering for grass at middle distance from the camera, where rendering of individual grass blades is too expensive due to their number. Our approach allows real-time rendering and a good parallax effect, which is important when the camera moves: the grass seems to have a real 3D shape and not flat as with billboards. The terrain to be rendered is divided into cells using a uniform grid. Over each cell, we lay a volume containing several thousands blades of grass. The volume width and depth correspond to the cell dimensions and its height is determined by the height of the tallest blade of grass. We then repeat this volume over the terrain. The way we minimize the presence of the repetitive pattern is explained in Section 3.1. Generally, methods using hardware 3D acceleration resort to slicing, where several slices of the volume are rendered using a 3D texture. A classic approach is to make the polygon planes facing the camera [17]. These polygons are semi-transparent to allow the visibility of the polygons behind and to define the global shape of the objects inside the volume. With our instancing approach, since the polygon coordinates are depending on the camera position, every coordinate would have to be computed for each polygon of every visible instance of the volume. This approach is too CPU and GPU intensive because of the linearly interpolated 3D texture accesses. So we use 2D slices, aligned with the three axes (middle of Figure 2). The geometry representing these slices is then static for any movement of the camera and the textures mapped onto these slices are 2D rather than 3D, thus faster to read. The use of slices for the three axes offers a good parallax effect, giving an illusion of depth. Moreover, there are no visible gaps between the slices since there are always slices on the two other axes that fill these gaps. Because of the vertical nature of grass blades, using multiple horizontal slices gives very poor results: many holes are visible between the slices, in particular when seen from a low altitude. Hence we use only one horizontal slice, close to the bottom of the patch to make it visible only when grass is seen from a high viewpoint.

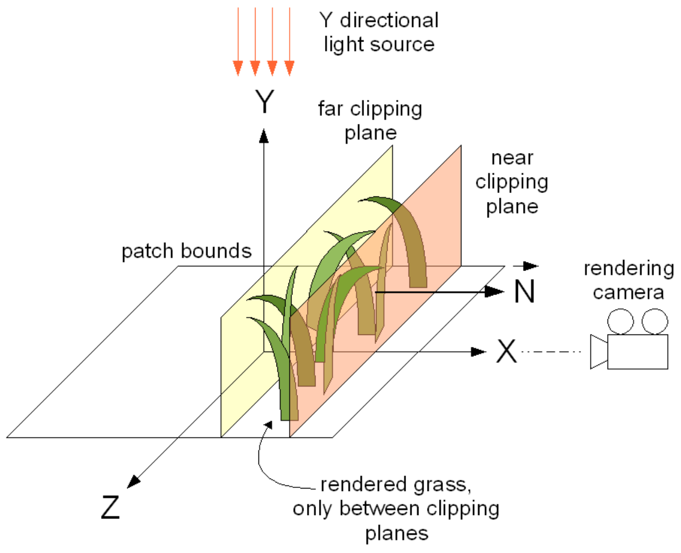
Y directional light source
Y
far clipping plane
near clipping plane
patch bounds
N
rendering camera
X
Z
rendered grass, only between clipping planes

Figure 6: Rendering of a slice orthogonal to the X axis. The current orthogonal camera is in front of the slice ($X+$). The current light direction is Y. The normal to the slice is $\vec{N}$ and determines the front face of the slice. The near and far clipping planes allow the rendering of only the grass blades needed for the current slice.

One of our goals is to render dynamically lit grass. We need per pixel lighting for our volume rendering approach since we do not define vertices for each blade of grass. We define the slice textures using semi-transparent BTFs rather than simple 2D images as for classic billboards. BTFs are originally dedicated to the representation of macro-structures on a surface (see Section 2), so they can represent small variations of height with simulation of self-shadowing and self-occlusions. BTF is a 6-dimensional function defining the reflectance at each point of a surface for any incidence and reflection direction. To reduce the amount of memory required to store the BTF data, we use a discrete representation of this function using a low number of light and view directions, 5 in our case: $Y+$, $X+$, $Z+$, $X-$ and $Z-$. $Y-$ is not defined because we consider that the camera and the light source cannot be under the ground. As the slices are axis-aligned and of zero thickness, we use only 2 view directions among the 5. Their choice depends on the slice directions. Due to the zero thickness, the slices are invisible from the remaining 3 directions. For example, only the $X+$ and $X-$

view directions are useful for the slices orthogonal to the X axis (as in Figure 6). An example data set for a slice orthogonal to the X axis is shown in Figure 7. Even though the sampling is very low (five directions), the results using these slices are visually pleasing for grass surfaces with diffuse only reflectance.

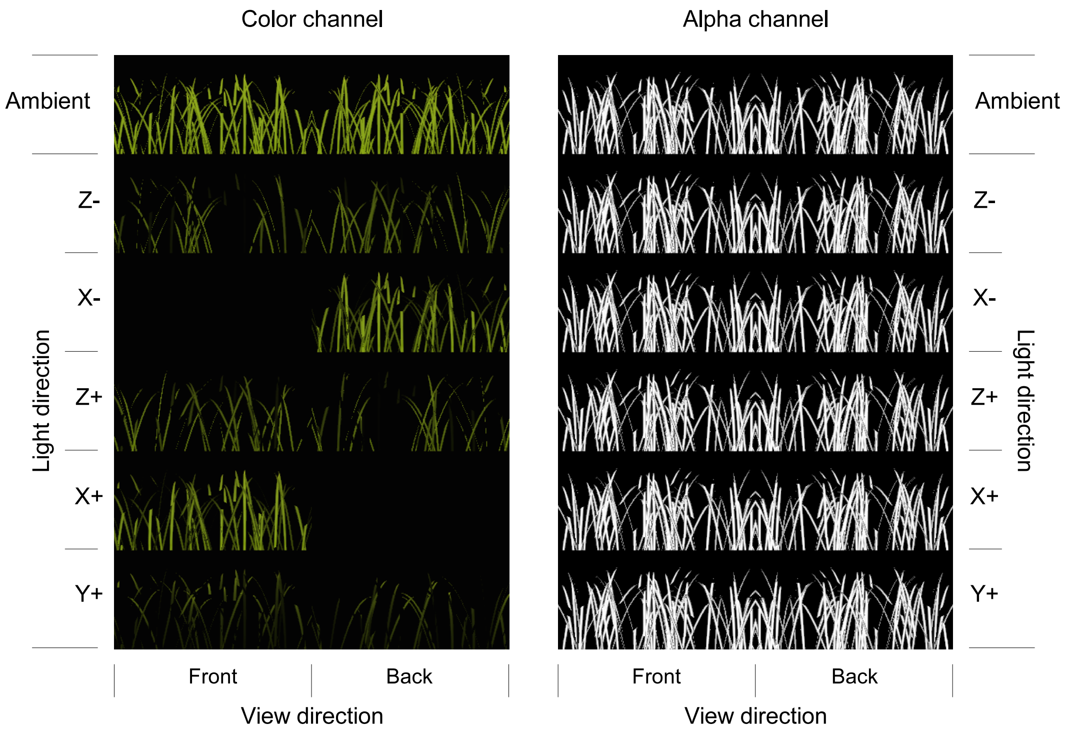

Figure 7: BTF data for a slice orthogonal to the X axis. The left image is the color channel, the right one is the alpha channel of the texture that stores a representation of the BTF. The five light directions are stored along the vertical axis, the view directions along the horizontal axis. Front view of a slice corresponds to watch it from the same side the slice normal is pointing to. A sixth row represents the diffuse reflectance factor of the grass material that is used to account for ambient lighting.

To create the slices data, we resort to a method similar to Meyer [14] for volume textures creation: a section of a patch of grass defined with geometry is rendered between two clipping planes. In our approach, we perform this process for each slice, each light direction and each view direction. This process is fast, a few seconds, since we use geometry rendering using 3D hardware acceleration. This pre-processing step has to be done only once since the resulting slices data are stored on the hard drive. The method for a slice orthogonal to the X axis is illustrated in Figure 6. Light direction is one of the BTF parameters, not light position. Thus we use a directional light source to illuminate the grass blades. For the same reason, an orthographic camera is used in place of a perspective one. Two clipping planes orthogonal to the camera direction vector are needed. We use the near and far clipping planes originally used for the Z-buffer range limits. These planes are located at the bounds of the currently rendered slice.

To create the alpha channel of the BTF, for semi-transparent volume slices, we use the above method (Figure 6) but using geometry with a constant white color on a black background and using the alpha channel of the blade texture (Figure 5*(c)*). We obtain grayscale images (right in Figure 7) with small gradients at the grass blade borders. The gradients allow anti-aliasing when performing the final rendering. The alpha channel is independent of the light direction, thus can be rendered once for a given view direction. To create the part of the slice data used for the ambient component computation (top left of Figure 7), we render a patch using geometry without any lighting computation, only the color of the grass blades appear.

As mentioned earlier, we render large grass fields by tiling elementary volumes. In a given volume, there are parts of grass blades whose roots are in the neighbor volumes. Thus, cut blades can appear between rendered patches if the BTF data are computed using only one patch of geometric grass. To handle this problem, we render the desired patch and its eight neighbors when creating the BTF data.

When rendering the grass field, we use the Lambert model. For a single omnidirectional light source applied to a point of a surface:

$$I = I_{ambient} + I_{diffuse} = K_d I_a + K_d \; max(\vec{N} \cdot \vec{L}, 0) \; \frac{I_d}{1+\beta d^2}$$

where K_d is the diffuse reflectance factor of the material, I_a the intensity of the ambient light, I_d the intensity of the point light source, d the distance between the surface point and the light source, β the attenuation factor, $\vec{N}$ the normal to the surface, $\vec{L}$ the light direction from the surface point, and I the rendered pixel intensity. The 1 in the denominator avoids an infinite intensity when being very close to the light source. Some of these values are taken from the sampled BTF. Only 5 light directions are available so we need to interpolate to get the inbetween BTF values, otherwise there would be sudden changes of color when moving the light source. We use a spherical barycentric interpolation of the BTF images (see Appendix A for detail). In the following equation, we compute the diffuse part by combining the images of the directions L_i, $i \in \{1..5\}$.

$$\begin{aligned} I &= K_d I_a + \left(\sum_{i=1}^{5} \alpha_i K_d \; max(\vec{N} \cdot \vec{L_i}, 0) \right) \frac{I_d}{1+\beta d^2} \\ &= K_d I_a + \left(\sum_{i=1}^{5} \alpha_i C_i \right) \frac{I_d}{1+\beta d^2} \end{aligned}$$

The K_d term is taken from the first row of the BTF image (Figure 7). This term is also used for the final ambient component computation. The $K_d \; max(\vec{N} \cdot \vec{L_i}, 0)$ term is taken from the five next rows of the image (called C_i in appendix A) and is used for the diffuse component computation. α_i are the interpolation coefficients, computed as in Appendix A. Two of these coefficients are 0 since only three samples are considered for a given quadrant of the hemisphere of possible directions. For a directional light source (the sun for example), the coefficients are computed once for the whole grass surface. Lighting computation due to a point light source requires per pixel computation of the previous equation and hence is very expensive. Per vertex computation of the α_i coefficients with per pixel linear interpolation provides a good cost/quality compromise. We interpolate only the diffuse component of the light and separately manage the ambient component as the ambient light value is constant for every point of the grass field.

A BTF is a function depending on the light and view directions. We interpolate sampled BTF data according to the incident light direction. For the view direction, interpolation is not useful since only two directions are defined and represent each face of the volume slices. If the camera is on the side where the slice normal vector is pointing, the front face has to be rendered (left column of Figure 7). Otherwise, the back face is rendered (right column of Figure 7).

For grass that is very far from the camera, particularly present for viewpoints at high altitude, we use a part of our volume rendering approach: only the horizontal slices are rendered. This approach looks similar to classic 2D texturing. However, our approach allows per pixel lighting and semi-transparency to see the ground beneath. Depending on the view angle and the distance from the grass, the visibility of the ground varies. The best results are obtained using anisotropic filtering. However, trilinear filtering is usually enough and allows better performances but a slightly lower rendering quality.

3.4 Density management and seamless transitions

In nature, grass is never uniformly distributed over the ground. Various external phenomena introduce chaos: kind of ground, availability of water [6], rocks, roads, etc., hence affecting the grass *density*. We define grass density as the number of grass blades by unit of surface. The density of grass at a point of the ground, which we call *local density*, can be defined with a *density map* covering the terrain (left of Figure 8).

It is often difficult to manage grass density in real-time applications. Thus, when many instances of a primitive, a tree for example, have to be distributed on a terrain using a density map, the position of each instance is computed in a preprocessing step. These positions are then used to translate each instance of the primitive when achieving the rendering of the final scene. In the case of grass, the required memory to store all grass blade locations is unavailable for large terrains containing hundreds of million grass blades. We define a method that does not require the preprocessing step and the storage space to locate the instances.

A user defined density map (left of Figure 8) gives information to create arbitrary distributions of grass (right of Figure 8) on the terrain. This map is equivalent to a probability distribution function. We use bilinear interpolation to get the local density for each point of the ground. A simple way to simulate different values of grass density would be to change the opacity of an uniform distribution of grass blades depending on the local density. However, the results do not look natural. We want to keep the full opacity of the grass blades and change the number of rendered grass blades while keeping the global uniformity of the blades distribution.

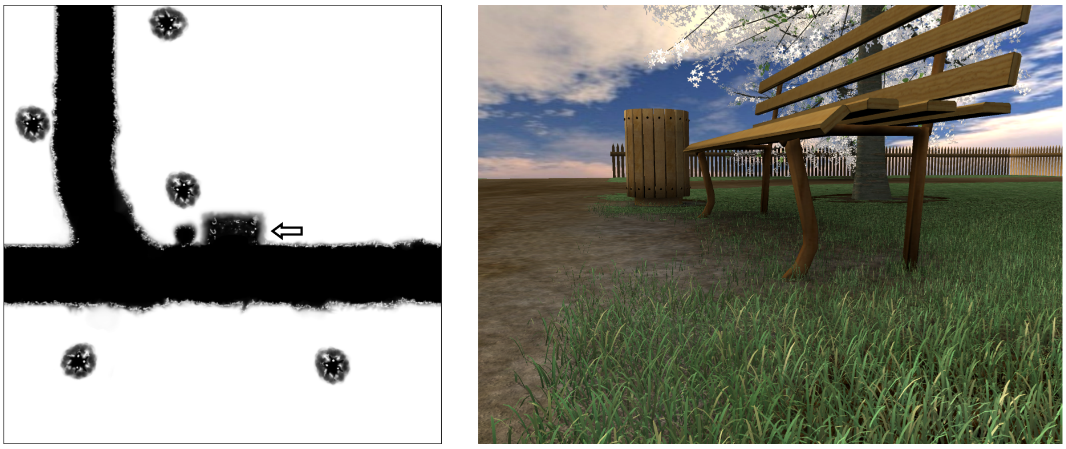

Figure 8: Modeling of grass using a bilinearly interpolated density map. The left image is an example of density map to render the scene of the right image. The arrow represents the camera position. Black pixels represent regions with no grass, white pixels represent the regions with maximum density.

We recall that a base grass patch is repeated over the terrain. However, we want the rendering of these patches to be different depending on the local density defined by the density map. Thus we introduce the notion of *density threshold* (Figure 9*(a)*). With each blade of the base grass patch is associated a threshold value ranging in $]0,1]$. During the rendering step, for each blade of each rendered patch, a test is performed before rasterization: if the blade density threshold is greater than the local density taken from the density map, then the blade is eliminated (Figure 9*(b)*). The blades with a high threshold are then rendered only in locations with high density. To keep the uniformity of grass distribution for any density value, we define the density thresholds randomly using an uniform distribution inside the single patch. To manage different species of grass, we can use a density map for each of the species and render the final scene in multiple passes.

Density for volume rendering has to be handled a different way since the images defining the BTFs for each slice do not carry information per blade of grass. Hence, we provide an additional texture per slice (Figure 10). Texels on this texture covered by a grass blade are assigned a value in $]0,1]$, stored as gray levels. Every texel belonging to a same grass blade should have the same gray level. To generate this image for each slice, we use the same method as that of the BTF generation: we render each grass blade of a patch between two clipping planes with a constant gray level proportional to the density threshold. At the time of actual rendering, the value from the density threshold channel is compared with the local density taken from the density map (Figure 11). If this density threshold is greater than the local density, the fragment is discarded. In Figure 11, the density threshold (0.8) at point B is greater than the local density (0.4) at the projected point B', so the pixel is not displayed. Conversely, the pixel A is displayed because its density threshold (0.2) is lower than the local density (0.3).

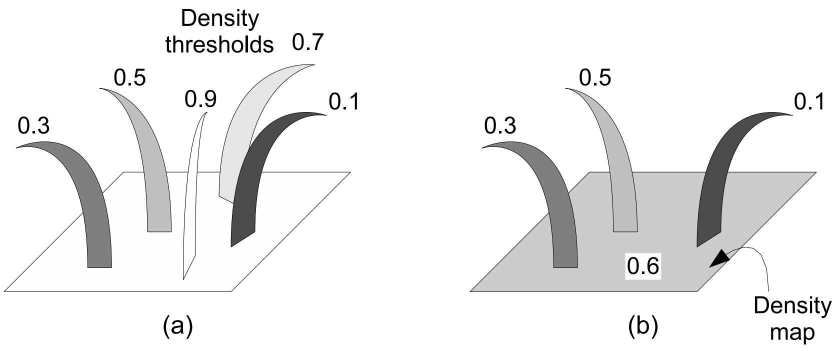

Figure 9: Density thresholds management. *(a)* With each grass blade is associated a density threshold in $]0, 1]$. *(b)* Rendering of this patch of grass using a density map with a constant value of 0.6. Only blades with a threshold lower or equal to 0.6 are rendered.

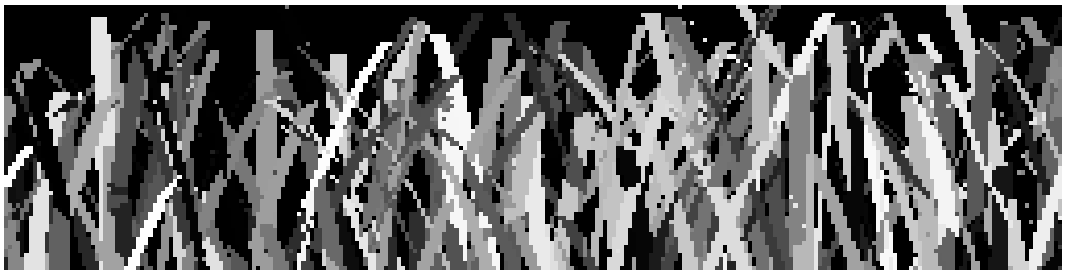

Figure 10: Density thresholds for one slice.

Our LOD scheme combines both geometry-based and volume-based rendering to render large grass terrains in real-time. However, transitions are visible from a method to another one. Smooth transitions between the levels of detail are desirable (Figure 12). A simple approach consists in fading from a rendering method to the other one by progressively changing the opacity depending on the distance from the viewer. The result does not look natural due to the presence of semi-transparent blades.

We propose to use our density management to perform seamless transitions. In the region between two levels of detail, the two rendering methods are used at the same time but with different densities, depending on the distance from the viewer. Hence, a subset of the grass blades is rendered with geometry, the remaining blades are rendered with volume slices. We use weight functions depending on the distance from the viewer, shown in Figure 13. For each grass blade processed at rendering time, we multiply the local density taken from the density map by the weight function corresponding to the current rendering method. Then, we perform the comparison with the blade density threshold.

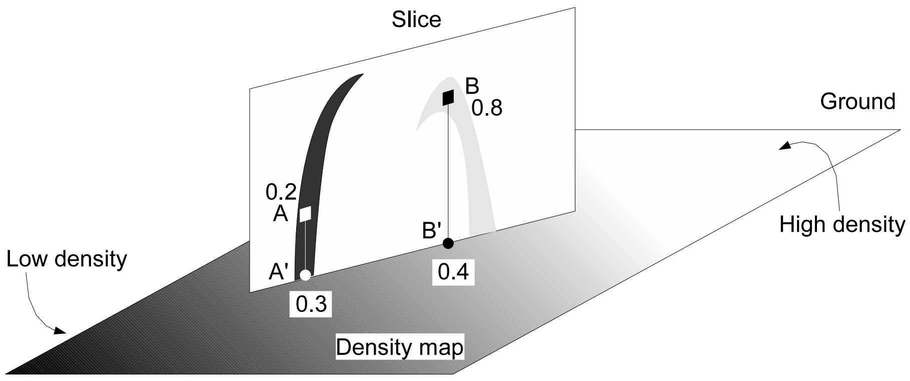

Figure 11: Rendering of a slice using the density threshold per pixel. A and B are the pixels to be tested. Their projections onto the ground are A' and B'. With the given density values, A is displayed but not B.

The different used weight functions are defined hereafter. The parameters *minGeom*, *maxGeom*, *minBTF*, *maxBTF* (see Figure 13) are defined by the user, depending on the desired rendering quality. The higher these parameters, the higher the rendering quality but at a higher cost. We define d as the distance from the camera, $w_g(d)$ the weight function for grass defined by geometry. We also define the $clamp(x, min, max)$ function as following:

$$clamp(x,min,max) = \begin{cases} min & \text{if } x \leq min, \\ x & \text{if } min < x < max, \\ max & \text{if } x \geq max \end{cases} \tag{1}$$

We define the weight function for geometry as following:

$$w_g(d) = clamp\Big(\frac{d - maxGeom}{minGeom - maxGeom}, 0, 1\Big) \tag{2}$$

For the vertical slices, the weight function has two slopes:

$$w_{vs}(d) = clamp\Big(\frac{d - minGeom}{maxGeom - minGeom}, 0, 1\Big).clamp\Big(\frac{d - maxBTF}{minBTF - maxBTF}, 0, 1\Big) \tag{3}$$

The function for the horizontal slices keeps only the first slope of the previous function:

$$w_{hs}(d) = clamp\Big(\frac{d - minGeom}{maxGeom - minGeom}, 0, 1\Big) \tag{4}$$

Problems appear when applying exactly the same comparison between the local density multiplied by the weight function and the density threshold for the two rendering methods. For example, at a distance from the viewer where the weight for the geometry is 0.4 and the weight for volume

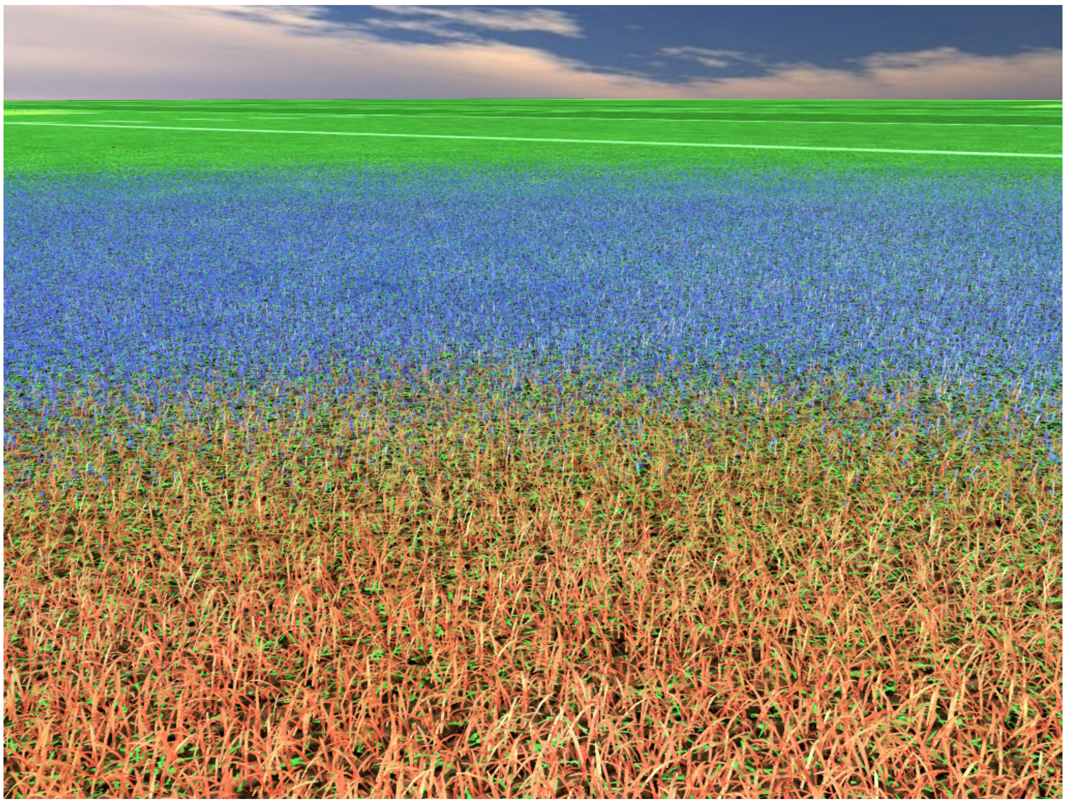

Figure 12: Smooth transitions between levels of detail made visible with false colors. Grass blades rendered with geometry are red. Vertical slices are blue. Horizontal slices are green.

slices is 0.6, geometric blades with a density threshold in]0,0.4] and blades on slices with a density threshold in]0,0.6] are rendered. So, the grass blades with a density threshold in]0,0.4] are rendered twice, and the ones in]0.6,1] are not rendered. To solve this problem, we use $1 - densityThreshold$ rather than $densityThreshold$ for blades defined by geometry when performing comparison with the local density. Therefore, grass blades on slices are rendered if $densityThreshold \in]0,0.6]$, while grass blades defined by geometry are rendered if $(1-densityThreshold) \in]0.6,1]$. In this case, there is no duplicated grass blades anymore.

When rendering grass with variable density depending on a density map, blades of grass are either rendered or not. However, using this method to manage transitions between levels of detail creates slightly visible popping artifacts when moving the camera. Rendering a grass blade or not is equivalent to setting its opacity to 1 or 0 respectively. This behavior is shown in Figure 14*(a)* where a grass blade is rendered if the local density multiplied by the weight function is greater than its density threshold dth. We would like to have a smooth transition, as in Figure 14*(b)*. Hence the

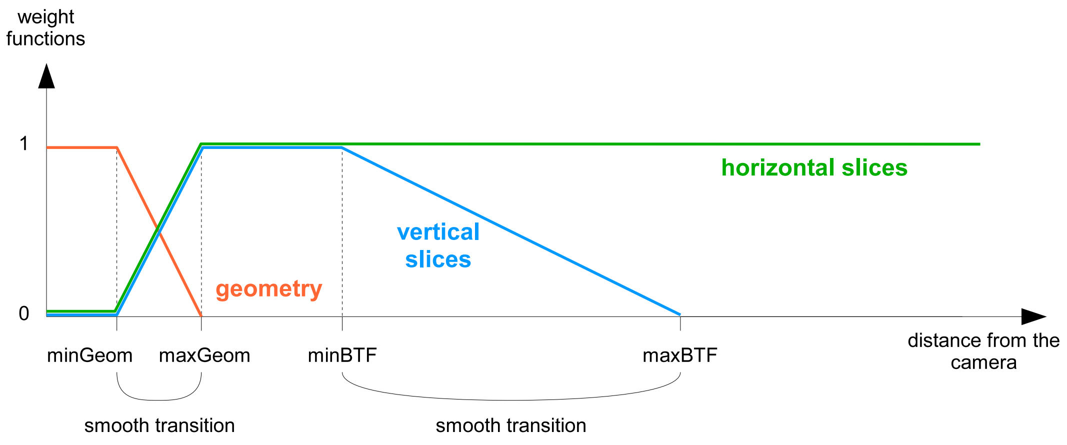

Figure 13: Functions weighting the local density for each rendering method, depending on the distance to the viewer. *minGeom*, *maxGeom*, *minBTF* and *maxBTF* are user-defined parameters.

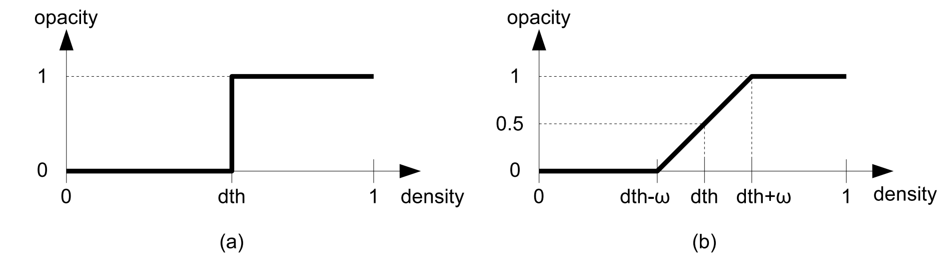

Figure 14: Opacity of a grass blade depending on the local density and the density threshold *dth*. *(a)* Simple function, a grass blade is either rendered or not. *(b)* Function providing a smoother transition.

definition of the following function:

$$opacity(density, dth) = clamp\left(\frac{density - dth + \omega}{2\omega}, 0, 1\right) \tag{5}$$

where ω is equal to half of the width of the transition region, 0.4 is a value that works well.

3.5 Shadows

Shadows are an important factor of realism in rendered scenes. If they are not present, the rendered images look flat, with low contrast, and it is difficult to know the exact location of 3D objects relatively to the others. However, rendering scenes with shadows involves expensive computations, resulting in low frame rates. If we render exact shadows for each grass blade, the computation cost gets prohibitive. We need to perform fast approximations that give visually pleasant dynamic shadows. There are three kinds of shadow: points of the ground occluded by grass blades (Figure 15), points of grass blades occluded by other blades (Figure 16) and self-shadowing of the blades. We do not manage the latter because these shadows appear rarely due to the shape of grass blades. We use a different algorithm for each kind of shadow.

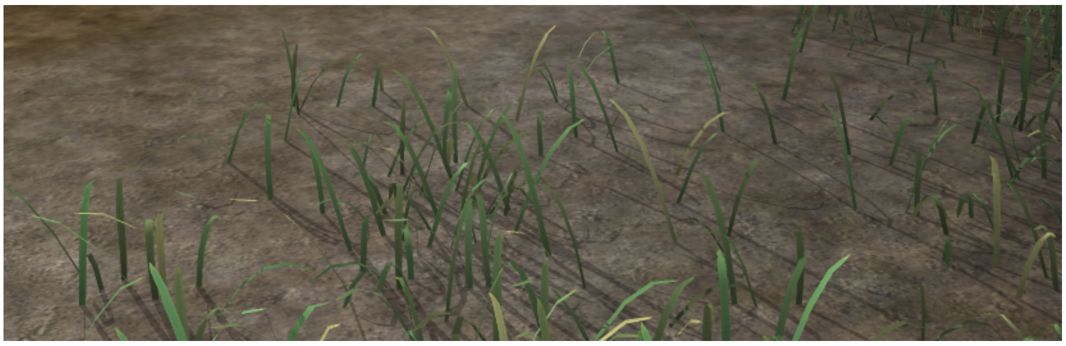
Figure 15: Shadows projected onto the ground.

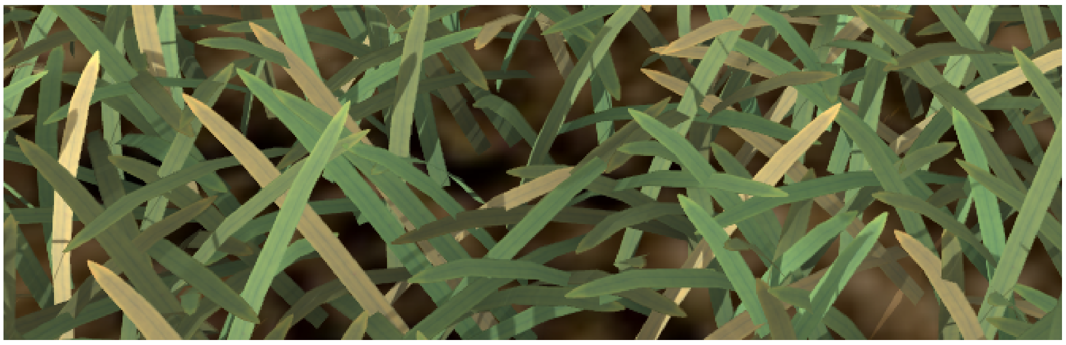
Figure 16: Shadows due to neighbor grass blades.

To render geometric grass blade shadows onto the ground, we use a classical projection method. A projection matrix is computed depending on the ground plane equation and the light source position. It transforms grass blade vertices to shadow vertices at the ground level. The stencil buffer is cleared, then the grass blades are rendered into the stencil buffer using the previous projection matrix and without lighting computations. At the end of this rendering, the stencil buffer contains boolean values indicating the location of shadows. We then render a black quadrilateral with blending covering the screen, making then the shadows visible. This approximative method gives correct results but only hard shadows can be rendered. For farther grass, using volume slices, we use a similar method: we project only the horizontal slice of each patch onto the ground using the same projection matrix. No stencil buffer is needed for this operation since the slice shadows do not overlap. We do not project the vertical slices, otherwise the projected shadows would cover the whole ground and consequently would get invisible.

The projection method cannot be applied to shadows cast by grass blades onto other blades, because the shadow receivers are not planar. Algorithms dedicated to arbitrary shaped surfaces should be used. For instance, any shadow mapping technique uses a depth map from the light source viewpoint. However, it cannot be applied to grass since the resolution of the shadow map texture should be extremely high, otherwise strong aliasing artifacts occur. Shadow volumes consists in the projection of the shadow caster silhouettes, creating shadow volumes rendered into the stencil buffer. It cannot be applied to grass for real-time applications since the rasterization work is time demanding (the silhouette of each blade should be projected).

Therefore, we propose an approximation that allows real-time performances and that creates convincing anti-aliased shadows, without being exact. Rather than projecting neighbor blades onto each blade, which is too expensive, we simulate the presence of these neighbors. We define a *shadow mask*, a grayscale texture representing the occlusions due to the neighborhood of a single grass blade (Figure 17). During the rendering of a blade of grass, a cylinder is fit around it with the origin of the shadow mask texture always aligned with the inclination direction of the blade (X axis in Figure 17). The shadow mask is computed once and for all and is used for each rendered blade of grass. It is a coarse approximation but gives convincing results for a real-time application. To create a shadow mask, a slice of a patch is rendered using an orthographic camera and a white background. Note that the rendered blades are assigned a black color.

At rendering time, for each vertex of a grass blade, a ray is launched from this vertex to the light source and the intersection point with the surrounding cylinder is computed. These coordinates are then interpolated for each pixel of the grass blade. For a given point on the blade, if the ray intersection point is in the range of the cylinder bounds, the corresponding shadow mask texel is retrieved. Then it is multiplied by the incoming light intensity to obtain the real incoming light intensity. Bilinear interpolation of the shadow mask provides anti-aliased shadows, shown in Figure 16.

In case of volume rendering, shadows cast by blades onto other blades can be straightforwardly handled. Indeed, the BTF images are generated while using the shadow mask algorithm. Then, at rendering time, no additional cost is implied since the volume rendering algorithm does not change.

A natural scene is never made of grass blades only. Additional elements in a scene, called *external elements*, could act as occluders for grass blades. To account for these occlusions, we use an *ambient occlusion* technique. For a point of the ground, a hemisphere centered at this point covers the possible incident light directions. The surface of this hemisphere corresponding to directions with occluders, divided by the area of the whole hemisphere is called ambient occlusion. Since we define this value for each point of the ground, we provide the information through an *ambient occlusion map* covering the terrain (Figure 18). This map is computed once and for all in a preprocessing step since it is independent of the point light source direction (only the ambient light is concerned). When rendering a grass blade, the ambient light intensity is multiplied by the value read from the ambient occlusion map at the root of the blade.

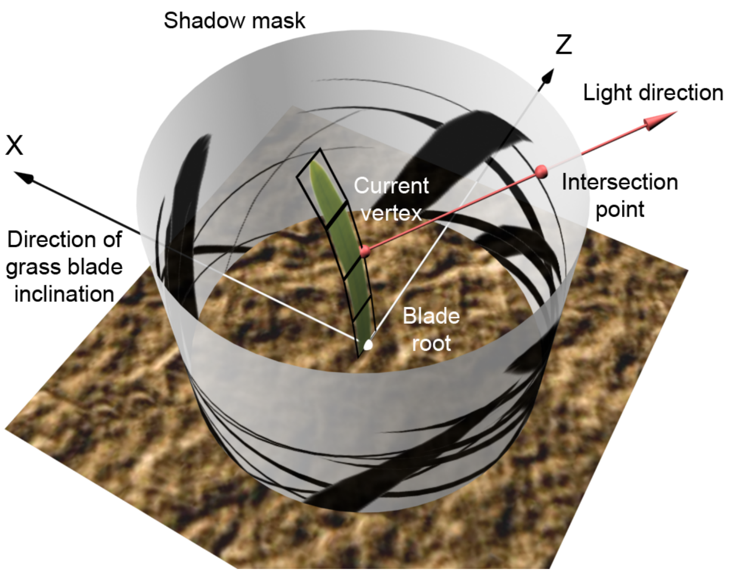

Figure 17: Rendering step: projection of shadows coming from the neighbor grass blades. A shadow mask mapped onto a cylinder is used as a visibility function.

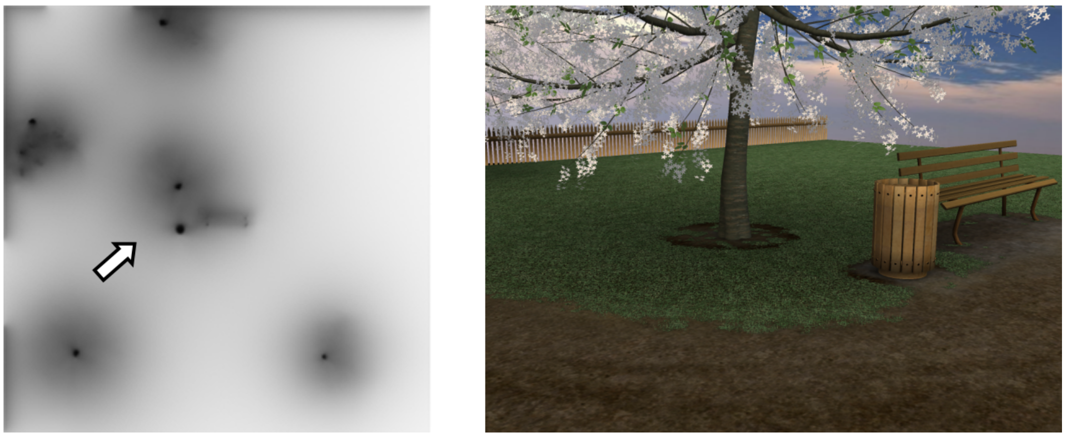

Figure 18: Ambient occlusion map covering the terrain and example of rendering. The arrow represents the camera position to get the image on the right.

4 Implementation hints

When implementing our grass rendering method, we come up with many issues, particularly due to filtering, mipmapping and aliasing. Speed issues have also to be solved due to the large size of usual terrains. We start by describing our multi-resolution terrain management, making our levels of detail approach efficient. Then we present solutions to the filtering problems, offering flicker free rendering of grass at any distance with a satisfying speed.

4.1 Multi-resolution approach for the terrain

In our approach, the terrain supporting grass is represented by an uniform grid. However, if viewed from faraway, an excessive number of cells has to be rendered. Direct frustum culling of the uniform grid is too expensive for large terrains. To reduce the culling and rendering times, we use a *quadtree* structure: a tree is initially built, each leaf contains the bounding sphere of a cell of the grid, the upper nodes contain the bounding spheres of square groups of cells, called *macro-cells*. When rendering a frame, the nodes to be processed are determined depending on the camera position, the camera frustum and the distance from the bounding sphere to the camera. The farther the cells from the camera, the larger the macro-cells to be rendered (Figure 19). Consequently, the total number of rendered cells is decreased.

The smallest cells are close to the viewer. Patches defined by geometry and the ones using vertical slices are rendered only in these cells. For grass faraway from the camera, only horizontal slices are rendered, particularly in the macro-cells. In these macro-cells, we use texture repetition to simulate a higher number of grid cells at the initial uniform grid resolution. The original texture coordinates for a macro-cell ($\in [0,1]^2$) are multiplied by its size in terms of number of cells.

Section 3.1 presented the way aperiodic tiling is performed to mask repetition patterns over the terrain. A random symmetry for each grid cell is used. However it cannot be applied to macro-cells by simply repeating texture coordinates. We define a *patch orientation map* (Figure 20*(a)*), a texture mapped over the terrain where each texel corresponds to a patch in a cell. This texture is not filtered so its value is constant along a grass patch. In the red channel of the map is stored a value of -1 or 1 representing the symmetry factor for the X axis. The green channel contains -1 or 1 for the Z axis. If symmetries are not used (Figure 20*(b)*), the coordinates (u', v') used to access the horizontal slice texture are defined as following:

$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} (patchSize\ .\ u) \bmod 1 \\ (patchSize\ .\ v) \bmod 1 \end{pmatrix} \tag{6}$$

where $patchSize = 4$ in Figure 20. To manage symmetries (Figure 20*(c)*), we propose the following modification of the previous equation:

$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} (patchSize\ .\ orientationMap(u,v).red\ .\ u) \bmod 1 \\ (patchSize\ .\ orientationMap(u,v).green\ .\ v) \bmod 1 \end{pmatrix} \tag{7}$$

With this equation, u' and v' are always in $[0,1]$ and vary according to the patch orientation map.

Rotations could have been applied rather than symmetries. For 1×1 cells, it would have been simple. However, they are more expensive to compute for each pixel of each rendered horizontal slice in the macro-cells. The Equation 7 is much simpler than that corresponding to rotations.

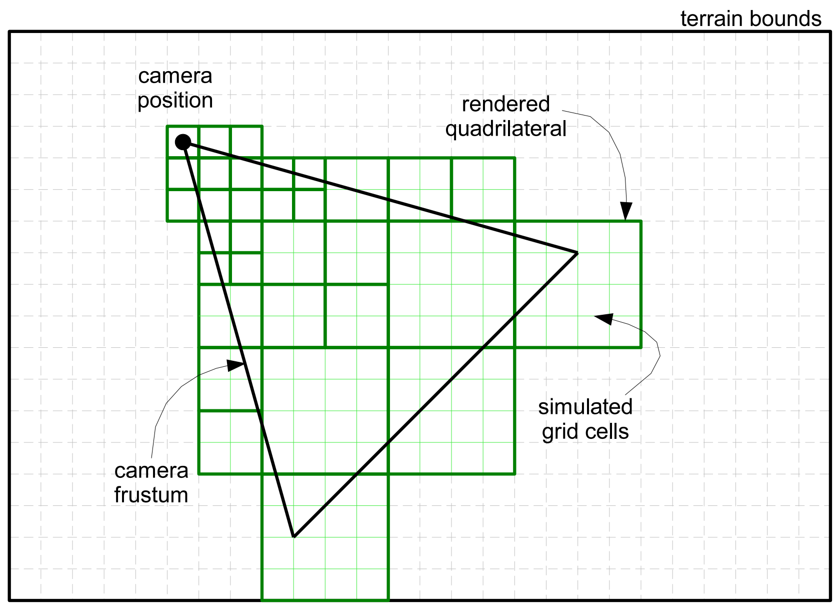

Figure 19: Terrain managed using a quadtree. The dark green quadrilaterals are rendered, determined by the camera position and orientation. The simulated grid cells are created using texture repetition.

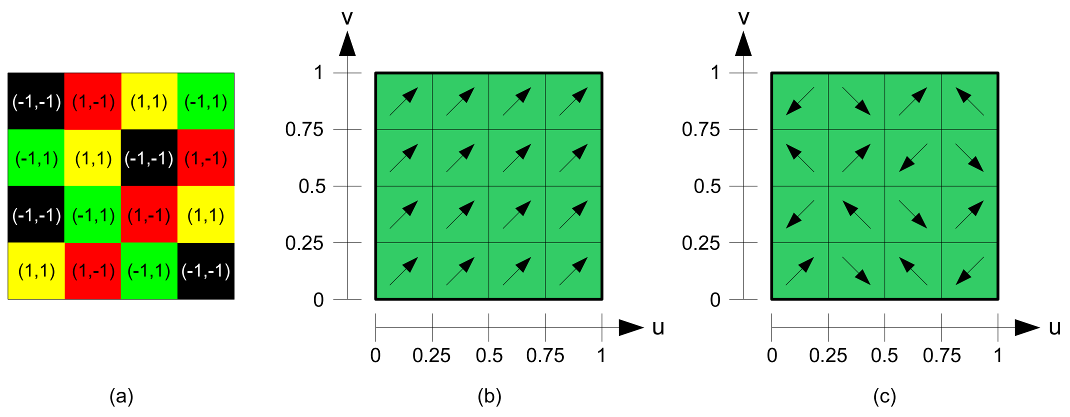

Figure 20: *(a)* Subset of a patch orientation map covering a 4×4 cell. *(b)* 4×4 macro-cell without symmetries management. The repetition of the grass patches is visible. *(c)* Symmetries applied to the grass patches, removing the repetition effect.

4.2 Order-independent rendering of semi-transparent quadrilaterals

The rendering order of several semi-transparent quadrilaterals is crucial when the aim is to avoid visual artifacts. Due to the high number of primitives (geometric grass blades, slices), the sorting is a process that is too expensive. Blending mixes the currently rasterized fragment with the color already stored in the current pixel. The main drawback of using blending alone is the importance of order: the blades have to be displayed from back to front, otherwise wrong pixels appear as in Figure 21*(a)*. Sorting of grass blades has to be done by the CPU each time the camera is moving, thus requiring an prohibitive processing time. Conversely, alpha test requires no sorting because the process is binary: the fragment is rendered only if its alpha value fulfills a condition. There are no partially transparent pixels depending on the pixels behind. The most important problem of alpha test is the aliasing as in Figure 21*(b)*. Our approach makes use of both blending and alpha test (Figure 21*(c)*). To eliminate the fragments outside a blade, we use alpha test with a low threshold. Therefore we obtain a coarse version of the blade shape. Blending refines the transparency process by mixing the borders with the background pixels, creating then an anti-aliasing effect. Blending artifacts still occur. However, only on the one pixel wide borders, which are invisible in most cases.

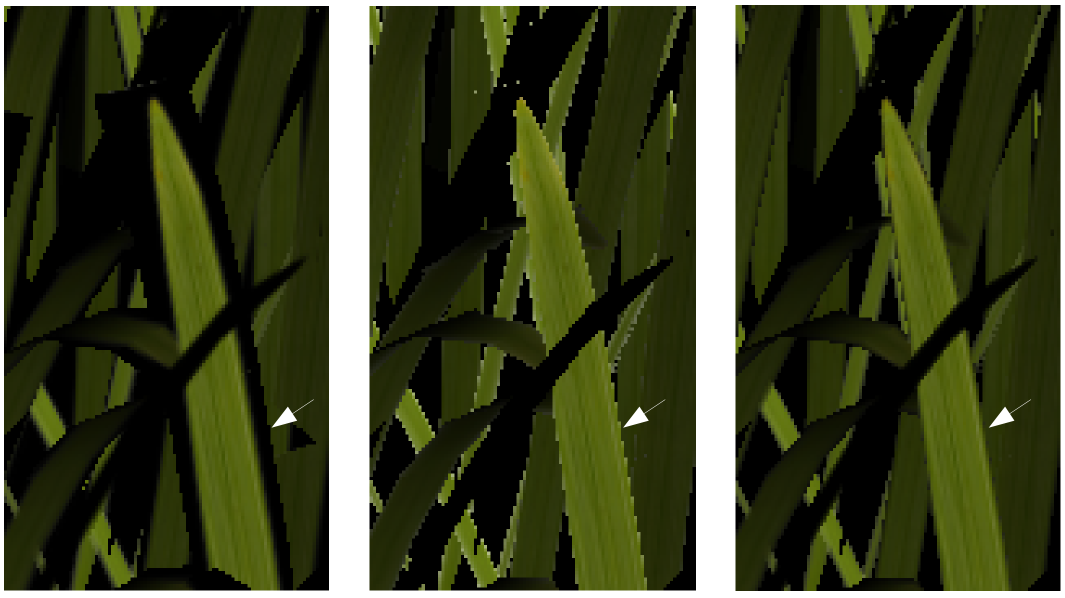

Figure 21: Comparison between three methods to process semi-transparency. The white arrows point to a zone where the advantages and drawbacks particularly appear. *(a)* Simple alpha blending. Sorting for rendering has to be done. *(b)* Alpha test, with a high aliasing effect. *(c)* Both methods, reducing the aliasing and the need for sorting.

The blending method we propose greatly decreases the aliasing due to alpha testing. It can be further decreased. However the needed feature is powerful enough only on latest graphics hardware since it requires a high fillrate. We make use of *multisampling* and enable the feature that maps the alpha value of the fragments to coverage value. No sorting of the grass blades is needed, wrong borders totally disappear and grass blades do not exhibit aliasing. Nevertheless, a high precision of the multisample buffer is needed to be able to remove the dithering effect (at least 6X). The choice of the method depends on the type of graphics card and the user's preference for quality or speed. The results we show in the figures of this report make use of multisampling.

5 Results

Our levels of detail approach allows the rendering of large grass terrains in real-time. Our implemention using the *OpenGL Shading Language* for the shaders, on a 3.2 GHz Pentium IV with *nVidia GeForce 7800 GTX*, renders a football field with about 627 million virtual grass blades (left of figure 1) at a rate of 18 to 250 frames per second (fps) in 1024×768 with 4X anti-aliasing. The rendering speed varies in this range as a function of the camera position and orientation.

Here is the summary of the rendering speeds we obtain with two different demos:

demo	football field (figure 23)	park scene (24)
whole terrain	250 fps	150 fps
human height	100 fps	80 fps
horizontal view at low altitude	18 fps	27 fps

Our levels of detail management allows high frame rates for faraway views and for the view at human height. The rendering is slowest when the vertical slices cover a large surface of the rendered window. That case happens when grass is observed horizontally from low altitude. The BTF slices rendering speed depends on the number of rasterized fragments due to the high depth complexity (several fragments are processed per pixel). In the case of a football game, this kind of view is not really useful.

The generation of slices data takes about 5 seconds for the 21 slices used in the park demo. 10 vertical slices are defined for each horizontal axis of a 0.5 meter wide patch. The resolution of the images for vertical slices is 512×64 and 512×512 for the horizontal slice. The total amount of data is 45 megabytes without any compression. Only 2 seconds are needed at run-time to load the BTF data.

An interesting advantage of our levels of detail scheme is the possibility to render a virtually infinite number of grass blades, as long as the data structures fit into memory. We have successfully rendered 25 times more grass blades than the football field (a total of 11 billions) with no variation of speed for close view, and a decrease from 250 to 200 frames per second for faraway view due to the larger screen coverage by the larger terrain.

Dynamic lighting is usually difficult to be obtained in 3D applications due to the needed computations. Our approach allows dynamic lighting and shadowing. The light position can be changed as desired, as shown in Figure 22.

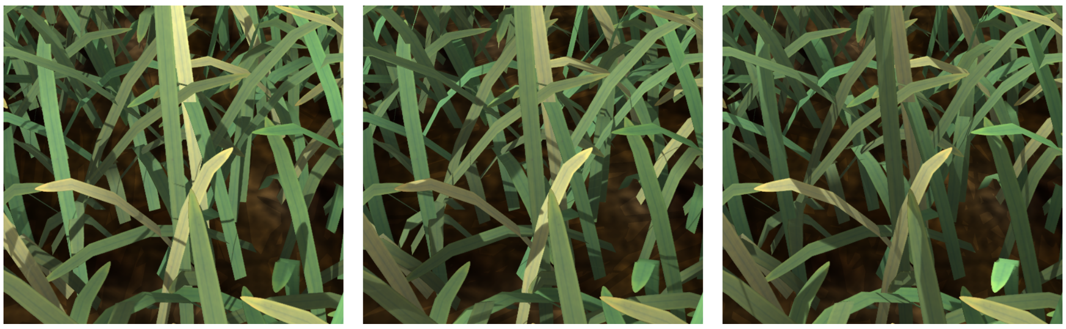

Figure 22: Results of dynamic lighting. A point light source is rotating around the grass surface. *(left)* Light source at the bottom of the image. *(middle)* Light source on the left. *(right)* Light source at the top of the image.

6 Conclusion

The objective of our work was to render dynamically lit, shadowed, anti-aliased grass with density management in real-time. Our approach mixes geometry, volume and surface rendering. The geometry rendering is used for rendering grass in the proximity of the camera, volume rendering for moderate distances and surface rendering for distant grass. Our main contribution is our volume rendering algorithm: we discretize an unit volume into slices, and for each slice we compute a BTF, a specific texture taking illumination, self-shadowing and self-occlusions into account. Each BTF allows us to get the reflected luminance to the viewer given an incident light direction. We discretize the space of light and view directions to compute only a limited subset of the BTFs and obtain a fast approximation. The inbetween light and view directions are computed using spherical barycentric interpolation. In addition to our contribution to the handling of dynamic lighting and shadows, we introduce the notion of density to create arbitrary shaped surfaces of grass. It also allows to dynamically manage the seamless transitions between the three rendering methods depending on the position and direction of the camera.

This work opens up a number of new directions of research. Our work is extensible to other natural elements: trees, plants, flowers, etc. It can also be extended several ways. A first example is BTF compression. This allows a higher sampling of light and view directions and reduces memory consumption, simplifying the memory management in presence of several species of grass. A second example is the management of curved terrains, introducing changes of coordinate frame for each part of the algorithm. Animation could also be added, typically by using wind simulation. However, the grass blades have to be long enough to make this animation useful.

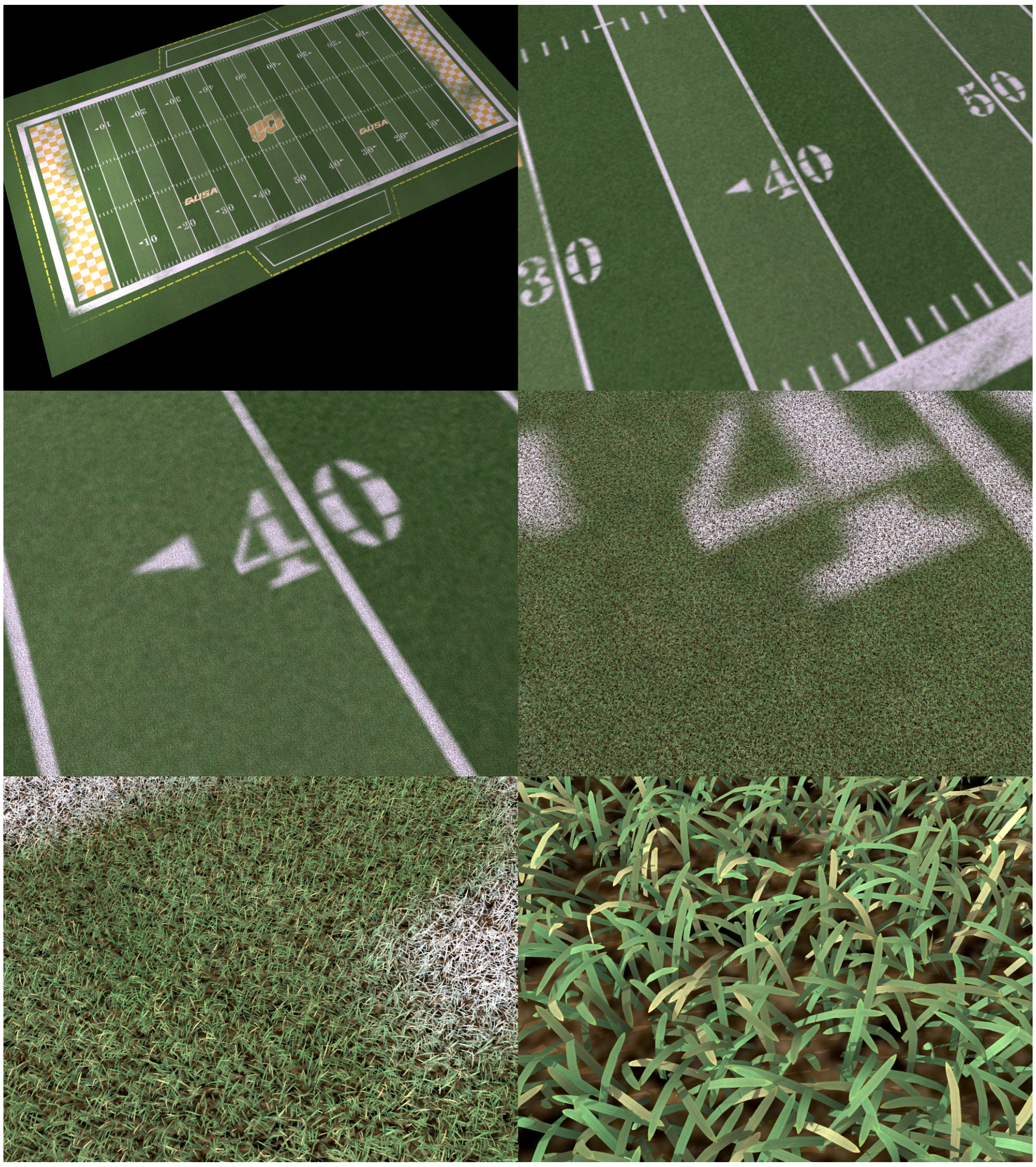

Figure 23: Football field seen at different distances.

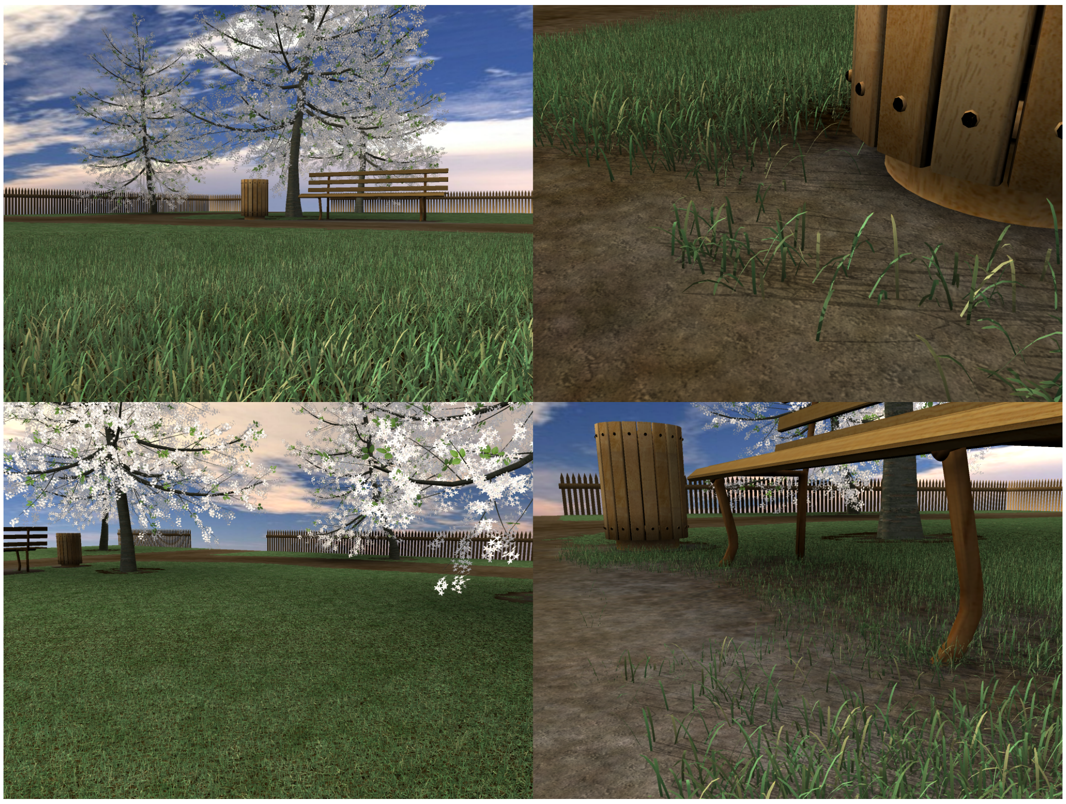
Figure 24: Park scene, using a density map to define grass distribution.

A Appendix

The discretization of the BTF representation for slices is very coarse: five light and five view directions. If we simply choose the correct texture on each slice depending on the light and view directions, sudden changes of intensity can be observed, hence the need for interpolation. The parameters we define are presented in Figure 25. The vector $\vec{l} = \overrightarrow{OL}$ represents the light direction. In this example, we have to interpolate between the images corresponding to the $X+$, Y and $Z+$ directions (A, B and C points). We propose a linear interpolation with spherical barycentric coordinates defining the weights. This interpolated color C_{int} for a rendered pixel is defined as following:

$$C_{int} = \alpha_x\, C_{X+} + \alpha_y\, C_Y + \alpha_z\, C_{Z+} \tag{8}$$

with C_{X+}, C_Y, C_{Z+} the colors taken from the corresponding images, α_x, α_y and α_z the barycentric coordinates in $[0,1]$. These three coordinates are proportional to the area of the three spherical triangles LBC, LCA and LAB. $\mathscr{A}(LAB)$ is the area of the spherical triangle LAB for example. $\mathscr{A}(ABC) = \frac{\pi}{2}$ is the area of the quarter of hemisphere ABC, of radius $R = 1$.

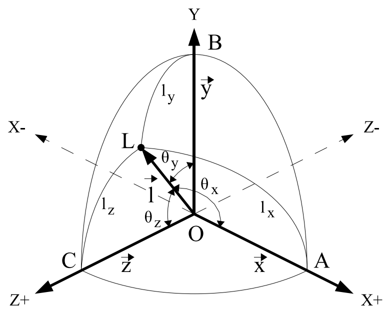

Figure 25: Parameters for the interpolation of slices images depending on the light direction $\vec{l}$

$$C_{int} = \frac{\mathscr{A}(LBC)}{\mathscr{A}(ABC)}\, C_{X+} + \frac{\mathscr{A}(LCA)}{\mathscr{A}(ABC)}\, C_Y + \frac{\mathscr{A}(LAB)}{\mathscr{A}(ABC)}\, C_{Z+} \tag{9}$$

To determine the area of triangle LBC, for the computation of α_x, we have:

$$\mathscr{A}(LBC) = \widehat{LBC} + \widehat{BCL} + \widehat{CLB} - \pi \tag{10}$$

The cosines of the angles of equation 10 can be determined using the following equations, for a unit sphere:

$$\begin{cases} \cos\widehat{LBC} = \dfrac{\cos\theta_z - \cos\theta_y \cos\frac{\pi}{2}}{\sin\theta_y \sin\frac{\pi}{2}} = \dfrac{\cos\theta_z}{\sin\theta_y} \\ \cos\widehat{BCL} = \dfrac{\cos\theta_y - \cos\theta_z \cos\frac{\pi}{2}}{\sin\theta_z \sin\frac{\pi}{2}} = \dfrac{\cos\theta_y}{\sin\theta_z} \\ \cos\widehat{CLB} = \dfrac{\cos\frac{\pi}{2} - \cos\theta_y \cos\theta_z}{\sin\theta_y \sin\theta_z} = \dfrac{-\cos\theta_y \cos\theta_z}{\sin\theta_y \sin\theta_z} \end{cases} \tag{11}$$

This expression can be simplified using the following relations:

$$\left\{\begin{array}{l}\cos\theta_y = \vec{l}\cdot\vec{y} = l_y \\ \cos\theta_z = \vec{l}\cdot\vec{z} = l_z \\ \sin\theta_y = ||\vec{l}\times\vec{y}|| = ||(-l_z, 0, l_x)^T|| = \sqrt{l_z^2 + l_x^2} \\ \sin\theta_z = ||\vec{l}\times\vec{z}|| = ||(l_y, -l_x, 0)^T|| = \sqrt{l_x^2 + l_y^2}\end{array}\right. \tag{12}$$

To find the angles of equation 10, we take the arccosines of equations 11:

$$\begin{aligned}\mathscr{A}(LBC) = & \arccos\frac{l_z}{\sqrt{l_z^2+l_x^2}} + \arccos\frac{l_y}{\sqrt{l_x^2+l_y^2}} \\ & + \arccos\frac{-l_y l_z}{\sqrt{l_z^2+l_x^2}\sqrt{l_x^2+l_y^2}} - \pi\end{aligned} \tag{13}$$

We do the same process for the areas $\mathscr{A}(BCL)$ and $\mathscr{A}(CLB)$. Finally, we obtain the surfaces of the three spherical triangles, and are able to compute α_x, α_y and α_z of equation 8.

These coefficients are computed per vertex rather than per pixel to greatly reduce the overhead at the GPU level. This is possible if the slices are small enough and if the light source is far enough. Linear interpolation is done per pixel, but the sum of the coefficients is not equal to 1, so renormalization has to be performed per pixel. There are nine arccosines to compute per vertex. It looks intensive for the GPU, however the bottleneck of slices rendering is the fragment shading. So the introduction of these arccosines does not influence the final rendering speed.

We just have presented an example using $X+$, Y and $Z+$ images. Computations are almost the same for the three other quadrants of the hemisphere when the light vector is inside these quadrants.

References

[1] 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. ACM, July 1985.

[2] 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*. ACM, Eurographics, ACM Press, 2001.

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

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

[5] Musawir A. Shah, Jaakko Konttinen, and Sumanta Pattanaik. Real-time rendering of realistic-looking grass. In *Proceedings of GRAPHITE '05*, pages 77–82, November 2005.

[6] 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. ACM, 1998.

[7] David Whatley. *GPU Gems 2*, chapter 1 - Toward Photorealism in Virtual Botany, pages 7–25. Addison-Wesley, March 2005.

[8] G. Müller, J. Meseth, M. Sattler, R. Sarlette, and R. Klein. Acquisition, synthesis and rendering of bidirectional texture functions. In Christophe Schlick and Werner Purgathofer, editors, *Eurographics 2004, State of the Art Reports*, pages 69–94. INRIA and Eurographics Association, September 2004.

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

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

[11] 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.

[12] 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. ACM, July 1989.

[13] 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.

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

[15] 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. ACM, March 2001.

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

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

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