

---


Electronic Theses and Dissertations, 2004-2019

---

2008

## Real-time Realistic Rendering Of Nature Scenes With Dynamic Lighting

Kevin Boulanger  
*University of Central Florida*

 Part of the [Computer Sciences Commons](#), and the [Engineering Commons](#)  
Find similar works at: <https://stars.library.ucf.edu/etd>  
University of Central Florida Libraries <http://library.ucf.edu>

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact [STARS@ucf.edu](mailto:STARS@ucf.edu).

---

### STARS Citation

Boulanger, Kevin, "Real-time Realistic Rendering Of Nature Scenes With Dynamic Lighting" (2008).  
*Electronic Theses and Dissertations, 2004-2019*. 3508.  
<https://stars.library.ucf.edu/etd/3508>

REAL-TIME REALISTIC RENDERING OF NATURE SCENES WITH  
DYNAMIC LIGHTING

by

KÉVIN BOULANGER

M.S. INRIA, University of Rennes I, France, 2005

A dissertation submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy  
in the School of Electrical Engineering and Computer Science  
in the College of Engineering and Computer Science  
at the University of Central Florida  
Orlando, Florida

Summer Term  
2008

Major Professor:  
Sumanta N. Pattanaik

© 2008 by K evin Boulanger

## ABSTRACT

Rendering of natural scenes has interested the scientific community for a long time due to its numerous applications. The targeted goal is to create images that are similar to what a viewer can see in real life with his/her eyes. The main obstacle is complexity: nature scenes from real life contain a huge number of small details that 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 goal of our research is to overcome this complexity and to achieve real-time rendering of nature scenes while providing visually convincing dynamic global illumination. Our work focuses on grass and trees as they are commonly visible in everyday life. We handle geometry and lighting complexities for grass to render millions of grass blades interactively with dynamic lighting. As for lighting complexity, we address real-time rendering of trees by proposing a lighting model that handles indirect lighting. Our work makes extensive use of the current generation of Graphics Processing Units (GPUs) to meet the real-time requirement and to leave the CPU free to carry out other tasks.

## ACKNOWLEDGMENTS

I want to thank my supervisors, Kadi Bouatouch and Sumanta N. Pattanaik, for the opportunity they gave to me to be in a joint supervision PhD program, and for their helpful comments about the research work and the whole process of paper publication. I also thank Charles E. Hughes and Hassan Foroosh for being in the PhD committee and for their help with the administrative process of the PhD program at the University of Central Florida. I thank Guillaume François, Musawir A. Shah, Jaakko Konttinen and Jonathan Brouillat for their numerous helpful suggestions. This thesis has been partially supported by the Florida High-Tech Council, Electronic Arts and INRIA in France.

# TABLE OF CONTENTS

<b>LIST OF FIGURES</b> . . . . .	<b>x</b>
<b>LIST OF TABLES</b> . . . . .	<b>xx</b>
<b>CHAPTER 1 INTRODUCTION</b> . . . . .	<b>1</b>
<b>CHAPTER 2 BACKGROUND</b> . . . . .	<b>4</b>
2.1 Lighting . . . . .	4
2.1.1 Basic radiometric quantities . . . . .	5
2.1.2 Light transport equation . . . . .	9
2.1.3 Light sources . . . . .	10
2.1.4 Visibility function and ambient occlusion . . . . .	18
2.1.5 Global illumination . . . . .	20
2.2 Material properties . . . . .	21
2.2.1 BRDFs . . . . .	22
2.2.2 SBRDFs and BTFs . . . . .	27

2.3	Volume rendering . . . . .	28
<b>CHAPTER 3 PREVIOUS WORK . . . . .</b>		<b>32</b>
3.1	Nature modeling and rendering . . . . .	32
3.2	Grass rendering . . . . .	33
3.2.1	Geometry-based rendering methods . . . . .	34
3.2.2	Image-based rendering methods . . . . .	37
3.2.3	Volume-based rendering methods . . . . .	41
3.3	Tree rendering . . . . .	45
3.3.1	Geometry-based modeling and rendering methods . . . . .	45
3.3.2	Image-based rendering methods . . . . .	47
3.3.3	Volume-based rendering methods . . . . .	49
3.3.4	Tree lighting . . . . .	50
<b>CHAPTER 4 RENDERING GRASS IN REAL-TIME WITH DYNAMIC LIGHTING . . . . .</b>		<b>53</b>
4.1	Introduction . . . . .	53
4.2	Our grass rendering method . . . . .	54
4.2.1	Levels of detail . . . . .	55
4.2.2	Geometry-based rendering . . . . .	59

4.2.3	Volume rendering . . . . .	61
4.2.4	Management of non-uniform distribution of grass . . . . .	67
4.2.5	Seamless transition between levels of detail . . . . .	71
4.2.6	Shadows . . . . .	76
4.2.7	Management of the terrain . . . . .	81
4.2.8	Animation . . . . .	89
4.3	Implementation . . . . .	91
4.3.1	Global grass rendering algorithm . . . . .	91
4.3.2	Order-independent rendering of semi-transparent quadrilaterals . . . . .	96
4.3.3	A custom filter to create the mipmaps pyramid of a semi-transparent texture . . . . .	98
4.4	Results . . . . .	99

**CHAPTER 5 RENDERING TREES IN REAL-TIME WITH INDIRECT LIGHTING . . . . . 105**

5.1	Introduction . . . . .	105
5.2	Overview . . . . .	107
5.2.1	Constructing the tree envelope . . . . .	107
5.2.2	Lighting environment . . . . .	108



5.2.3	Leaf materials . . . . .	109
5.2.4	Attenuation function . . . . .	111
5.3	Direct Lighting . . . . .	112
5.3.1	Light from the sky and the ground . . . . .	112
5.3.2	Directional Light Source . . . . .	113
5.4	Indirect Lighting . . . . .	115
5.4.1	Integration Scheme . . . . .	116
5.4.2	Evaluation of the sums . . . . .	119
5.4.3	Evaluation of the irradiance on neighbor leaves . . . . .	121
5.5	Shadows . . . . .	123
5.5.1	Shadows projected onto leaves by other leaves . . . . .	125
5.5.2	Shadows cast by trees . . . . .	133
5.6	Implementation . . . . .	137
5.7	Results . . . . .	144
<b>CHAPTER 6 CONCLUSION AND FUTURE WORK . . . . .</b>		<b>154</b>
<b>APPENDIX A SPHERICAL BARYCENTRIC INTERPOLATION OF THE GRASS BTF DATA . . . . .</b>		<b>157</b>
<b>APPENDIX B SIMPLIFICATION OF EQUATION 5.15 . . . . .</b>		<b>162</b>

**LIST OF REFERENCES . . . . . 166**

## LIST OF FIGURES

2.1	Irradiance is the radiant flux per unit area incident at a point of surface coming from a hemispherical solid angle. . . . .	6
2.2	Intensity is the radiant flux per unit solid angle $d\omega$ emerging from a point. . . . .	6
2.3	Radiance $L$ is the radiant flux per unit solid angle $d\omega$ and per projected unit area $dA_p$ , which is incident on, emerging from, or passing through a point of a surface in direction $\omega$ . . . . .	7
2.4	Vector and angle definitions for BRDF models. Incident light from direction $\omega_i$ is reflected on point $P$ of surface $dA$ to the direction $\omega_o$ . $N$ is the normal to the surface and $\omega_r$ the ideal reflection direction. . . . .	10
2.5	Vector and angle definitions for an area light source. Incident light from direction $\omega_i$ in unit solid angle $d\omega_i$ is emitted from the unit area $dA'$ at distance $\ PP_l\ $ from the surface centered in $P$ . $N'$ is the normal to the area light surface. . . . .	11
2.6	Vector and angle definitions for a point light source. The flux received by $dA$ due to the point light source at $P_l$ , in direction $\omega_i$ , at distance $R$ , is the flux emitted by the light source in the unit solid angle $d\omega'$ . . . . .	13

2.7	Example of latitude-longitude mapping for environment lighting using a hemisphere. The lower part of the environment sphere is set to 0 in this case. . .	16
3.1	Grass blades using geometry. (a) Using trajectories of particles. The dots represent the generated coordinates. (b) Using a cubic Hermite curve. (c) Using a chain of billboards. . . . .	35
3.2	Three kinds of animation of grass objects from [Pel04]. (a) Per cluster of grass objects. (b) Per vertex. (c) Per grass object. . . . .	40
3.3	Two kinds of billboard deformation. (a) Quadrilateral billboard with shifted top. (b) Strip of polygons with progressive deformation. . . . .	41
4.1	Park scene rendered in real-time using our grass level of detail scheme. . . .	54
4.2	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 used. . . . .	56
4.3	Result of aperiodic tiling. (a) Periodic tiling of a single grass patch. Notice the repetitive pattern at the center of the image. (b) Random mirroring of the grass patch instances to achieve aperiodic tiling. Repetitive pattern is no more noticeable. . . . .	57

4.4	<p>(a) Grass blades defined with textured semi-transparent quadrilateral strips.</p> <p>(b) Original scanned grass blade. (c) Color channel of the blade texture modified to remove the white border. (d) Alpha channel of the texture defining the shape of the blade. . . . .</p>	59
4.5	<p>Rendering of a slice orthogonal to the X axis. The current orthographic camera is in front of the slice (<math>X+</math>). The current light direction is <math>Y</math>. The normal to the slice is <math>N</math> 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. . . . .</p>	63
4.6	<p>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 shown along the vertical axis, the view directions along the horizontal axis. Front view of a slice corresponds to the camera being on 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. . . . .</p>	64
4.7	<p>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. . . . .</p>	69

4.8	Density threshold management. (a) Each grass blade is assigned 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 threshold less than or equal to 0.6 are rendered. . . . .	70
4.9	Handling of density threshold for the rendering of slices. (a) $A$ 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 $B$ is not. (b) Example of data for a slice. . . . .	71
4.10	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. . . . .	72
4.11	Functions weighting the local density for each rendering method, as a function of the distance to the viewer. $minGeom$ , $maxGeom$ , $minBTF$ and $maxBTF$ are user-defined parameters. . . . .	73
4.12	Distribution of grass blade opacities for each rendering method, as a function of the grass blade density threshold $dth$ . If the density threshold of a blade is greater than the local density $ld$ , it is eliminated (gray region). . . . .	74

4.13	Values of $\omega$ , the size of the transition region between rendering methods, as a function of the weight $w_{vs}(d)$ . $\omega_0$ is a user-defined offset to avoid a division by 0 when computing the opacity (Algorithm 1). We use $\omega_0 = 0.05$ in our implementation. . . . .	75
4.14	Shadows projected onto the ground. . . . .	77
4.15	Shadows due to neighbor grass blades. . . . .	77
4.16	Shadow rendering step: projection of shadows coming from the neighbor grass blades. A shadow mask mapped onto a cylinder is used as a visibility function.	80
4.17	Ambient occlusion map covering the terrain and example of rendering. The arrow represents the camera position to get the image on the right. . . . .	81
4.18	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. . . . .	82
4.19	(a) Subset of a patch orientation map covering a $4 \times 4$ cell. (b) $4 \times 4$ macro-cell without mirroring. The repetition of the grass patches is visible. (c) Random mirroring applied to the grass patches, removing the repetition effect. . . . .	84
4.20	Distortion of the patches over the terrain. A coordinate frame $(T, N, B)$ is defined for each point of the terrain, computed using the values at the current patch corners. . . . .	85

4.21	Vertical slices added at the top of hills. They appear depending on the angle between the terrain normal and the view vector and only for distances from the camera ranging from <i>maxBTF</i> to <i>maxSilhouettes</i> . . . . .	89
4.22	Generation of a height map and density maps from a mesh. For the height map, white represents the location of the highest points. Each density map is associated with a type of grass, white represents presence of grass. . . . .	90
4.23	Animation of the grass patches. (a) Base position. (b) Distortions driven by wind vectors <i>W</i> . . . . .	91
4.24	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 testing, with a high aliasing effect. (c) Both methods, reducing the aliasing and the need for sorting. . . . .	97
4.25	Screenshots of the park demo. The grass terrain is modeled using user-defined height and grass density maps. These images are rendered at 10 to 16 frames per second using a <i>nVidia GeForce 7800 GTX</i> graphics card. . . . .	100
4.26	Football field of 627 million virtual grass blades seen at different distances. . . . .	101
4.27	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. . . . .	103



5.1	(a) Photograph of a tree. (b) Rendering with our method. . . . .	106
5.2	(a) Parametrization of the envelope of a tree from a leaf's center point $P$ . (b) Equivalent set of volumes $V_j$ . . . . .	108
5.3	The different lighting components. From left to right: direct lighting due to the sky and the light reflected off the ground ( $L_{o_{sky}}$ ), direct lighting due to the sun ( $L_{o_{sun}}$ ), indirect lighting due to the sun ( $L_{o_{ind}}$ ) and sum of all components ( $L_o$ ). . . . .	110
5.4	Four possible functions to estimate the shape of the tree envelope for a leaf centered at $P$ . The (red,green,blue) lines represent the coordinate frame of the leaf. The cyan lines are the vectors $\omega_j$ scaled by $s_{max}(P, \omega_j)$ with $N_{dir} = 8$ . The distance of the gray surface from $P$ represents the $\widetilde{s_{max}}(P, \omega_l)$ distances for all possible $\omega_l$ . . . . .	115
5.5	Tree rendering (a) without indirect lighting, (b) with indirect lighting. . . . .	117
5.6	Subdivision of a volume $V_j$ into slices of radius $s_i$ and of thickness $\Delta s_j$ . The blue line is an example of reflected ray on a leaf of the slice (left inset), the red dashed line is an example of transmitted ray through a leaf of the slice (right inset). For all leaves inside $V_j$ , $\alpha_j$ is the constant angle between $\omega_l$ and $\omega_j$ . . . . .	119
5.7	Cube shaped shadow mask. The leaf centered at $P$ is shadowed based on the texture value at the intersection of the ray ( $P, \omega_l$ ) and the cube. . . . .	126

5.8	Unfolded cube map of the shadow mask. . . . .	127
5.9	Thresholding of the shadow mask. (a) Original shadow mask (one of the faces of the cube). (b)-(e) Processed shadow mask with different increasing threshold values. . . . .	128
5.10	Procedurally generated sphere of leaves. Each leaf is assigned a gray level, lighter when farther from the center. The six rendering cameras (for the six faces of the cube) are placed at the center of the sphere. . . . .	129
5.11	Concentric hollow spheres containing subsets of the neighbor leaves. . . . .	131
5.12	Unfolded cube map of the multi-layer shadow mask (only RGB channels are visible, see Figure 5.13 for the alpha channel). . . . .	132
5.13	The four layers of the shadow mask (one of the faces of the cube). (a) Red channel, with gray levels from 0 to $t_1/t_4$ . (b) Green channel, with gray levels from $t_1/t_4$ to $t_2/t_4$ . (c) Blue channel, with gray levels from $t_2/t_4$ to $t_3/t_4$ . (d) Alpha channel, with gray levels from $t_3/t_4$ to 1. . . . .	132
5.14	(a) Generation of the shadow map. (b) Projection of shadows on a rendered object. . . . .	135
5.15	(a) RGB channels of the shadow map (only the green channel is used for the $Y$ axis). (b) Alpha channel of the shadow map (1 = fully transparent, 0 = fully opaque). . . . .	136

5.16	Comparing our method to Monte-Carlo rendering. Light comes from the top right corner. Top row: our rendering. Bottom row: Monte-Carlo rendering with 256 rays per intersection point. (a) Direct lighting from the sky. (b) Direct lighting from the sun. (c) Indirect lighting from the sky, ignored in our case. (d) Indirect lighting from the sun. These four light components represent 24%, 48%, 6% and 22% respectively of the total radiance value. (e) Sphere of leaves with false colors for indirect lighting due to the sun, red for reflected light and green for transmitted light. . . . .	139
5.17	The same scene in two different lighting conditions. . . . .	147
5.18	Our lighting method is dynamic, allowing real-time changes of lighting conditions. . . . .	148
5.19	The appearance of trees depends mainly on translucency, attenuation and indirect lighting. . . . .	149
5.20	100 trees rendered at 27 fps with indirect lighting and shadows. . . . .	150
5.21	Sunset scene. The left part of the tree has a yellow tint due to the sun color. This yellow tint is scattered inside the tree. Indirect lighting does not reach the right part of the tree where a green color is visible. . . . .	151
5.22	Shadows cast by leaves onto other leaves. . . . .	152
5.23	Shadow cast by a tree. Note the sharpness of the shadow close to the trunk and its smoothness farther from the tree. . . . .	153

A.1	Parameters for the interpolation of slices images depending on the light direction $l$ . . . . .	159
B.1	$N_{P'}$ , $\omega_j$ and $\omega_l$ in spherical coordinates. . . . .	165

## LIST OF TABLES

- 5.1 Rendering speed with different light components and different numbers of trees 145

# CHAPTER 1: INTRODUCTION

Modeling and rendering of natural scenes has interested the scientific community for a long time due to its numerous applications. The modeling of natural elements and the study of optical behavior of aggregates of natural elements are particularly used for remote sensing, cartography and ecosystems studies. Studies about plant growth are sources of new structural representations of natural elements, using grammars with sets of rules. Rendering of natural scenes is studied for visualization of such scenes in different contexts: ecosystem simulations, data of remote sensing, 3D representations of cartography data, video games, flight simulators, etc.

Nowadays, computer generated images of natural scenes are getting more and more realistic. Indeed, the targeted goal is to create images that are similar to what the viewer can see in real life with his/her eyes. The main obstacle in achieving the target is complexity. Nature scenes from real life contain a huge number of small details that 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 or tree leaves for example, as a huge number of primitives such as triangles are required to accurately model them; lighting computation complexity is due to the multiple reflections of light over scene objects with complex materials.

Overcoming this complexity has been a challenging problem for many years. We address this problem in the context of grass and tree rendering. Algorithms have been developed to render grass and trees in real-time with coarse approximations. Algorithms have also been developed to render them with high quality, for example using raytracing. Even in this latter case, approximations have to be used to render within a reasonable processing time.

The goal of our research is to achieve real-time rendering of nature scenes while providing visually convincing dynamic global illumination. Our goal is to render large amounts of natural elements with approximations that reduce the rendering time while giving the convincing illusion of global illumination in dynamic scenes.

We focus our work on grass and trees since they are commonly visible in everyday life. We study these topics separately due to their totally different structure. Our work can be extended to any other natural element; nature rendering is a very vast topic. Our goal is to find new algorithms that are applicable to widely varied situations, so the extension to other natural elements will require only a small amount of work.

We are using the current generation of Graphics Processing Units (GPUs) that make use of vertex shaders, programs executed per vertex, and fragment shaders, programs executed per fragment. Our algorithms are designed to maximize the use of these computing stages and leave the CPU free to carry out any other types of simulation (related to nature or not).

This thesis is structured as follows: we start by giving some notions related to lighting, reflectance and rendering. These notions are useful to understand the subsequent chapters and are the basics used by our algorithms. The next chapter presents the state of the art on

nature rendering, grass rendering and tree rendering. Having a knowledge of what results have already been achieved gives us a better idea on what has still to be done. The two next chapters present our research work, related to real-time grass and tree rendering, with sections showing the results we obtain. Finally, we conclude and discuss of some possible future work in the domain.



## CHAPTER 2: BACKGROUND

In this chapter, we describe the main notions we are using in our grass and tree rendering algorithms. It also gives the notations used through the different chapters. We start by giving some notions about lighting models and material properties. We also give some background on volume rendering.

### 2.1 Lighting

In a given environment, light propagates, starting from light sources and bouncing on objects present in the scene. The description of this distribution of radiance makes possible rendering of 3D scenes viewed from an arbitrary camera. Rendering consists of evaluating the radiance along camera rays, starting from the camera position and going through the camera view plane. We give first some basics about radiometry that are used to introduce the light transport equation. We then present several light source models used in different situations. We then present the visibility function and how it affects lighting in a scene. We finish the section by presenting global illumination and why it is a computationally expensive problem.

### 2.1.1 Basic radiometric quantities

We start by giving a couple of definitions of basic radiometric quantities, explained in detail in the book of McCluney [McC94]. They will help explain the principles of rendering, particularly the light transport equation, detailed in the next section.

*Radiant energy*  $Q$  is the quantity of energy propagating onto, from, or through a surface of given area in a given period of time. We consider the particle nature of light, in which -radiant energy can be expressed in terms of the number of photons that are propagating onto, from, or through the surface of given area in a given period of time. Radiant energy is measured in Joules ( $J$ ).

*Radiant flux (or power)*  $\Phi$  is the rate of flow of radiant energy per unit time, i.e., the quantity of energy transferring through a surface or region of space per unit time. It is expressed in Watts ( $W$ ), or Joules per second ( $J.s^{-1}$ ), and is defined as follows:

$$\Phi = \frac{dQ}{dt} \tag{2.1}$$

*Irradiance*  $E$  is the radiant flux per unit area  $dA$  of a given surface, which is incident on or passing through a point in the given surface (Figure 2.1). All directions in the hemisphere around the surface point have to be included. It is expressed in Watts per square meter ( $W.m^{-2}$ ) and is defined as follows:

$$E = \frac{d\Phi}{dA} \tag{2.2}$$

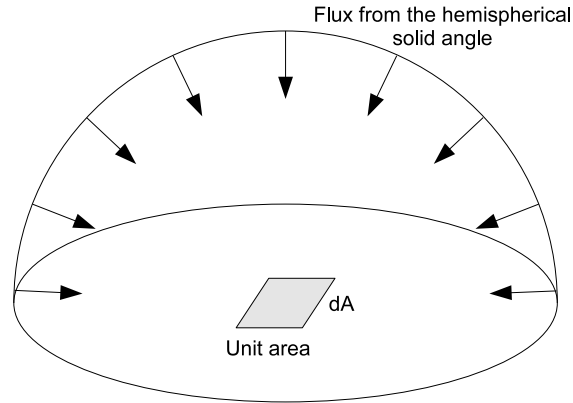


Figure 2.1: Irradiance is the radiant flux per unit area incident at a point of surface coming from a hemispherical solid angle.

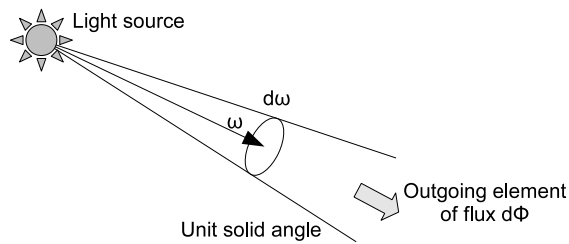


Figure 2.2: Intensity is the radiant flux per unit solid angle  $d\omega$  emerging from a point.

*Radiant exitance* is the same notion as irradiance, except that the energy leaves the surface rather than being incident to it. It is often written  $B$ .

*Radiant intensity*  $I$  is the radiant flux per unit solid angle  $d\omega$ , which is incident on, emerging from, or passing through a point in space in a given direction (Figure 2.2). It is expressed in Watts per steradian ( $W.sr^{-1}$ ) and is defined as follows:

$$I = \frac{d\Phi}{d\omega} \tag{2.3}$$

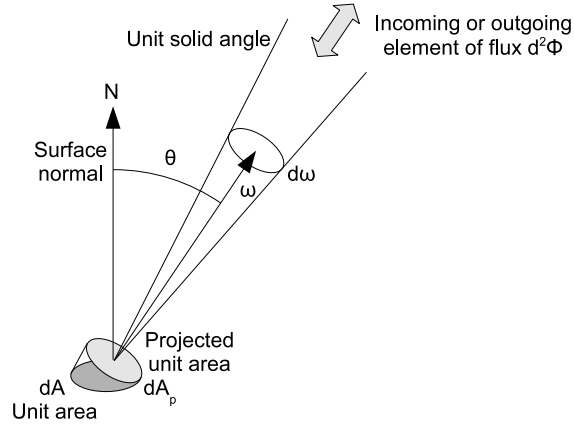


Figure 2.3: Radiance  $L$  is the radiant flux per unit solid angle  $d\omega$  and per projected unit area  $dA_p$ , which is incident on, emerging from, or passing through a point of a surface in direction  $\omega$ .

*Radiance*  $L$  is the radiant flux per unit solid angle and per projected unit area, which is incident on, emerging from, or passing through a given point of a surface in a given direction (Figure 2.3). It is expressed in Watts per square meter per steradian ( $W.m^{-2}.sr^{-1}$ ) and is defined as follows:

$$L = \frac{d^2\Phi}{d\omega dA_p} = \frac{d^2\Phi}{d\omega \cos\theta dA} \quad (2.4)$$

where  $dA_p = \cos\theta dA$  is the projected area, area of the surface orthogonal to the solid angle direction  $\omega$  and projected from the original element of area  $dA$ . The angle between the surface normal and the solid angle direction is called  $\theta$ .

Radiance is the main radiometric quantity that is used to describe light distribution in an environment. The other basic radiometric quantities can be defined in terms of radiance, which is integrated over a solid angle or an area.

Given incident radiance  $L_i(P, \omega_i)$  from direction  $\omega_i$  at a point  $P$  on a surface of normal  $N$ , we can compute the irradiance  $E(P)$  at the point  $P$  by summing the contribution of radiance from every direction in the hemisphere  $\Omega_+(N)$  oriented by  $N$ :

$$E(P) = \int_{\Omega_+(N)} L_i(P, \omega_i) \cos \theta_i d\omega_i \quad (2.5)$$

The cosine of the incidence angle,  $\cos \theta_i$ , is often defined as the dot product  $(\omega_i \cdot N)$ .

Spherical coordinates are a practical means to describe the direction vectors  $\omega_i \in \Omega_+(N)$  using two angles, the azimuthal angle  $\phi_i \in [0, 2\pi)$  and the elevation angle  $\theta_i \in [0, \pi/2]$ , which is the angle between  $\omega_i$  and  $N$ . The unit solid angle is expressed as  $d\omega_i = \sin \theta_i d\theta_i d\phi_i$ . The irradiance becomes:

$$E(P) = \int_0^{2\pi} \int_0^{\pi/2} L_i(P, \phi_i, \theta_i) \cos \theta_i \sin \theta_i d\theta_i d\phi_i \quad (2.6)$$

If  $L_i(P, \phi_i, \theta_i)$  is constant for every direction of the hemisphere and equal to  $L_i(P)$ , the irradiance becomes simple:

$$E(P) = \pi L_i(P) \quad (2.7)$$

Given outgoing radiance  $L_o(P, \omega_o)$  in direction  $\omega_o$  from a small surface around point  $P$  of normal  $N$  emitting light, we can compute the intensity  $I(P_l, \omega_o)$  at the center point  $P_l$  by

summing the contribution of radiance for each point of the emitting surface:

$$I(P_l, \omega_o) = \int_{A_l} L_o(P, \omega_o) \cos \theta_o \, dA \quad (2.8)$$

However, intensity is normally applied to points.  $A_l$  is actually considered very small compared to the distance of the observer from the light source.

### 2.1.2 Light transport equation

Before defining the light transport equation, we quickly present the notion of a Bidirectional Reflectance Distribution Function (BRDF). This function characterizes reflectance of a surface at a point (Figure 2.4). It is defined as the ratio of outgoing radiance over the incoming irradiance:

$$f_r(P, \omega_o, \omega_i) = \frac{dL_o(P, \omega_o)}{dE(P, \omega_i)} \quad (2.9)$$

We define it in more detail and present some models in Section 2.2.1.

The light transport equation (or rendering equation) describes the equilibrium distribution of radiance in an environment. It gives the amount of reflected (or outgoing) radiance  $L_o(P, \omega_o)$  from a unit surface  $dA$  centered at point  $P$  along reflection direction  $\omega_o$  given the irradiance  $dE$  from the environment at this point. It is obtained by integrating the product

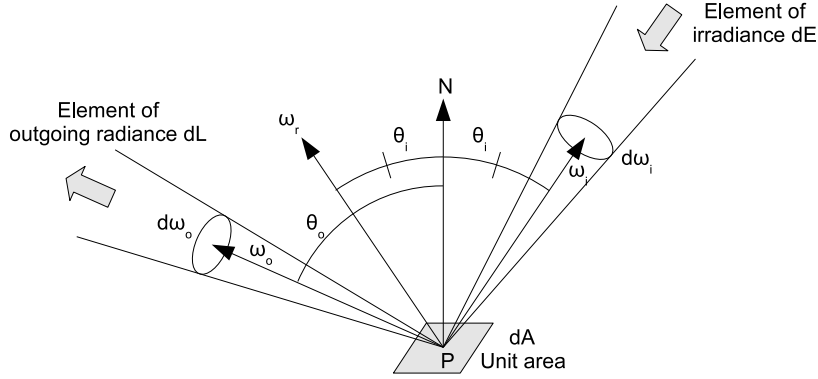


Figure 2.4: Vector and angle definitions for BRDF models. Incident light from direction  $\omega_i$  is reflected on point  $P$  of surface  $dA$  to the direction  $\omega_o$ .  $N$  is the normal to the surface and  $\omega_r$  the ideal reflection direction.

of the BRDF  $f_r(\omega_o, \omega_i)$  with the differential irradiance  $dE$ :

$$L_o(P, \omega_o) = L_e(P, \omega_o) + \int_{\Omega_+(N)} f_r(P, \omega_o, \omega_i) dE(P, \omega_i) \quad (2.10)$$

$$= L_e(P, \omega_o) + \int_{\Omega_+(N)} f_r(P, \omega_o, \omega_i) L_i(P, \omega_i) \cos \theta_i d\omega_i \quad (2.11)$$

where  $L_e(P, \omega_o)$  is the radiance due to self-emission,  $L_i(P, \omega_i)$  is the incident radiance and  $f_r(P, \omega_o, \omega_i)$  is the BRDF.

### 2.1.3 Light sources

Light sources are surfaces that emit light by themselves, they are only defined by their outgoing radiance. The outgoing radiance  $L_o(P_l, \omega_o)$  of a point  $P_l$  of a light source is then

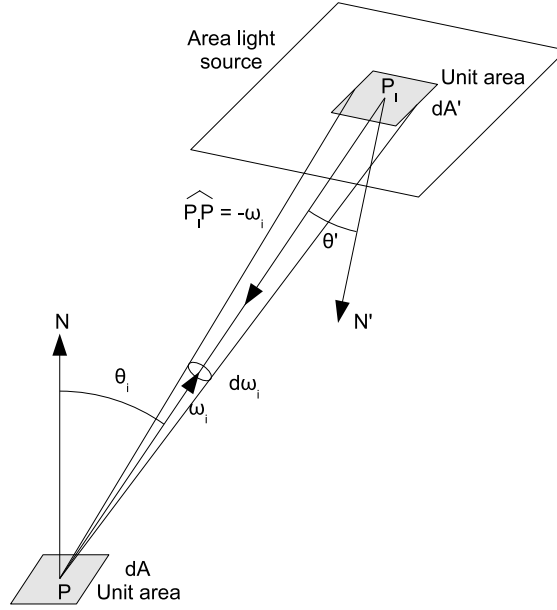


Figure 2.5: Vector and angle definitions for an area light source. Incident light from direction  $\omega_i$  in unit solid angle  $d\omega_i$  is emitted from the unit area  $dA'$  at distance  $\|PP_l\|$  from the surface centered in  $P$ .  $N'$  is the normal to the area light surface.

defined as:

$$L_o(P_l, \omega_o) = L_e(P_l, \omega_o) \quad (2.12)$$

If there is no participating media causing scattering along the ray  $P_l P$ , the radiance is constant along this ray. Therefore, we can modify the expression of the irradiance (Equation 2.5) and the expression of the light transport equation (Equation 2.11) using the following relation:

$$L_i(P, \omega_i) = L_o(P_l, -\omega_i) \quad (2.13)$$

## Area light sources

An area light source is a surface that emits light. To account for its contribution in the



irradiance and light transport equations, it is easier to integrate over the light source area rather than the subtended solid angle. In the previous section, irradiance and light transport equations are defined using the differential solid angle  $d\omega_i$ . We can express this solid angle using the unit area  $dA'$  using the following expression:

$$d\omega_i = \frac{\cos \theta' dA'}{\|PP_l\|^2} \quad (2.14)$$

where  $dA'$  is a unit area of the light source,  $\theta'$  is the angle between the light source surface normal  $N'$  and the vector from the light source to the surface being lit,  $\widehat{P_l P} = -\omega_i$  (Figure 2.5).

The irradiance at point  $P$  due to this light source becomes

$$E(P) = \int_{A'} L_i(P, \omega_i) \cos \theta_i \frac{\cos \theta' dA'}{\|PP_l\|^2} = \int_{A'} L_o(P_l, -\omega_i) \cos \theta_i \frac{\cos \theta' dA'}{\|PP_l\|^2} \quad (2.15)$$

and the light transport equation becomes

$$\begin{aligned} L_o(P, \omega_o) &= L_e(P, \omega_o) + \int_{A'} f_r(P, \omega_o, \omega_i) L_i(P, \omega_i) \cos \theta_i \frac{\cos \theta' dA'}{\|PP_l\|^2} \\ &= L_e(P, \omega_o) + \int_{A'} f_r(P, \omega_o, \omega_i) L_o(P_l, -\omega_i) \cos \theta_i \frac{\cos \theta' dA'}{\|PP_l\|^2} \end{aligned} \quad (2.16)$$

with  $A'$  the area of the light source surface.

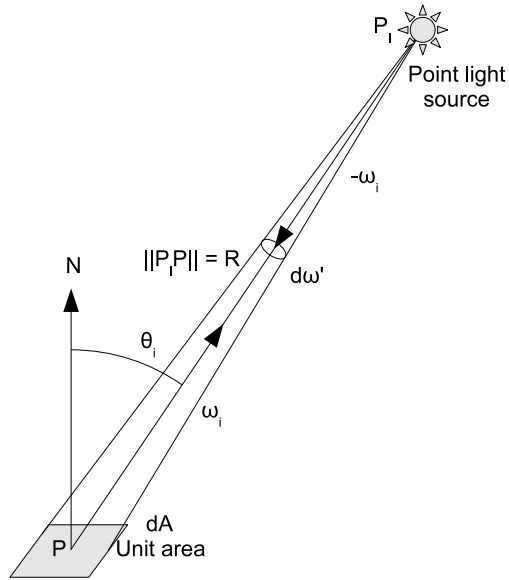


Figure 2.6: Vector and angle definitions for a point light source. The flux received by  $dA$  due to the point light source at  $P_l$ , in direction  $\omega_i$ , at distance  $R$ , is the flux emitted by the light source in the unit solid angle  $d\omega'$ .

### Omnidirectional point light sources

Point light sources are light emitters with a zero area. They are a practical way to approximate very small light sources since they greatly simplify the light transport equation evaluation. Omnidirectional point light sources emit a constant flux in every direction. A unit area  $dA$  centered in  $P$  receives the flux emitted by the light source in the unit solid angle  $d\omega'$  starting from the point light source position  $P_l$  and subtended by  $dA$ . Since the radiant intensity  $I$  represents the flux emitted by the light source per unit solid angle, the irradiance of the unit area  $dA$  due to this light source is

$$E(P) = \frac{I \cos \theta_i}{R^2} \quad (2.17)$$

Since flux is received by  $dA$  in only one direction,  $\omega_i = \widehat{PP_l}$ , the light transport equation (Equation 2.10) becomes

$$L_o(P, \omega_o) = L_e(P, \omega_o) + \int_{\Omega_+(N)} f_r(P, \omega_o, \omega_i) \delta(\omega_i - \widehat{PP_l}) dE(P, \omega_i) \quad (2.18)$$

where  $\delta(x)$  is the dirac operator defined as follows:

$$\delta(x) = 0 \quad \text{if } x \neq 0 \quad (2.19)$$

$$\int_D \delta(c - x) f(x) dx = f(c) \quad \text{if } c \in D \quad (2.20)$$

The light transport equation then becomes simple to evaluate:

$$L_o(P, \omega_o) = L_e(P, \omega_o) + f_r(P, \omega_o, \widehat{PP_l}) \frac{I \cos \theta_i}{R^2} \quad (2.21)$$

### Directional light sources

A directional light source can be represented as a collimated beam. In this case, light reaches surfaces in parallel beams. Irradiance  $E$  can be expressed for any virtual surface inside the beam. If the normal  $N$  of a surface makes an angle  $\theta_l$  with the beam direction  $\omega_l$ , the irradiance  $E(P)$  at any point  $P$  of this surface is  $E(P) = E_l \cos \theta_l$  with  $E_l$  the irradiance associated with the light beam. In this case, the light transport equation becomes

$$L_o(P, \omega_o) = L_e(P, \omega_o) + f_r(P, \omega_o, \omega_l) E_l \cos \theta_l \quad (2.22)$$

## Environment lighting

Environment lighting uses a light source at an infinite distance from the rendered scene and surrounding it completely. In this case, the light transport equation as defined in Equation 2.11 is used to compute the contribution of this light source, except if the incoming radiance from every direction is constant, in which case the light transport equation becomes trivial, the radiance for each incoming direction has to be determined. The most common approach is environment mapping. It uses an image (environment map or irradiance map) defining the radiance for a finite set of directions in the sphere of incoming light directions. Each pixel of the environment map can be represented as an area light source at an infinite distance, for which the constant outgoing radiance equals the value of the corresponding pixel. The light transport equation can then be written using the sum of the contribution of each light source where each light source  $k$  has a subtended solid angle  $\Omega_k$  from point  $P$ :

$$L_o(P, \omega_o) = L_e(P, \omega_o) + \sum_{k=1}^{N_{lights}} \int_{\Omega_k} f_r(P, \omega_o, \omega_i) L_i(P, \omega_i) \cos \theta_i d\omega_i \quad (2.23)$$

We made the hypothesis that the outgoing radiance is constant for each light source, hence

$$\begin{aligned} L_o(P, \omega_o) &= L_e(P, \omega_o) + \sum_{k=1}^{N_{lights}} f_r(P, \omega_o, \omega_{i_k}) L_i(P, \omega_{i_k}) \cos \theta_{i_k} \int_{\Omega_k} d\omega_i \\ &= L_e(P, \omega_o) + \sum_{k=1}^{N_{lights}} f_r(P, \omega_o, \omega_{i_k}) L_i(P, \omega_{i_k}) \cos \theta_{i_k} \Delta\omega_k \end{aligned} \quad (2.24)$$



Figure 2.7: Example of latitude-longitude mapping for environment lighting using a hemisphere. The lower part of the environment sphere is set to 0 in this case.

The solid angle  $\Delta\omega_k$  for each light source depends on the transformation from image space to world space of the environment map.

A common representation for environment lighting using an image is spherical environment mapping. The most common mapping from image space to world space is latitude-longitude mapping (Figure 2.7): a conversion from spherical to cartesian coordinates is performed to convert pixel positions to world space. For an image of  $N_c$  columns and  $N_r$  rows, the azimuthal angle  $\phi$  is mapped to the horizontal axis and the elevation angle to the vertical axis. The first and last rows represent the poles of the sphere. For a pixel of coordinates  $(i, j)$  with  $i \in 0..N_c - 1$  and  $j \in 0..N_r - 1$ , the corresponding spherical angles

are defined as follows:

$$\phi(i) = (i + 0.5) \frac{2\pi}{N_c} \quad \theta(j) = (j + 0.5) \frac{\pi}{N_r} \quad (2.25)$$

The solid angle associated with each pixel  $(i, j)$  is defined as follows:

$$\Delta\omega(i, j) = \frac{2\pi^2}{N_c N_r} \sin(\theta(j)) \quad (2.26)$$

The irradiance and lighting equations become discrete sums over the light sources:

$$E(P) = \frac{2\pi^2}{N_c N_r} \sum_{j=0}^{N_r-1} \sum_{i=0}^{N_c-1} Env(i, j) \max(\omega_i(i, j) \cdot N, 0) \sin(\theta(j)) \quad (2.27)$$

$$L_o(P, \omega_o) = L_e(P, \omega_o) + \frac{2\pi^2}{N_c N_r} \sum_{j=0}^{N_r-1} \sum_{i=0}^{N_c-1} f_r(P, \omega_o, \omega_i(i, j)) Env(i, j) \max(\omega_i(i, j) \cdot N, 0) \sin(\theta(j)) \quad (2.28)$$

where  $Env(i, j)$  is the value of the pixel  $(i, j)$ . The  $\max()$  function is used to take into account only one side of the current surface, the one for which the dot product between the incidence direction  $\omega_i(i, j)$  and the surface normal  $N$  is positive.

Environment lighting can be represented using spherical harmonics [RH01]. The outgoing radiance of the environment is projected into an orthogonal basis, giving a set of coefficients. The light transport equation (Equation 2.11) is in fact a scalar product between the surface BRDF and the irradiance. If both of them are expressed in the spherical harmonics basis, the scalar product between them is simply performed by the product of their spherical harmonics

coefficients. However, one of the BRDF or the irradiance spherical harmonics representations has to be rotated to be able to work in the same space. Ramamoorthi [RH01] claims that 9 coefficients are enough, making the evaluation of the light transport equation very efficient.

### 2.1.4 Visibility function and ambient occlusion

In previous sections, the light transport equation was evaluated considering no obstacle between light sources and the receiver surface. However, in normal scenes obstacles are always present, reducing the contribution of light sources for a set of incidence directions, hence creating shadows. We rewrite the light transport equation to take this shadowing into account:

$$L_o(P, \omega_o) = L_e(P, \omega_o) + \int_{\Omega_+(N)} f_r(P, \omega_o, \omega_i) V(P, \omega_i) L_i(P, \omega_i) \cos \theta_i \, d\omega_i \quad (2.29)$$

where  $V(P, \omega_i)$  is the visibility function, equal to 1 when a light source is visible and equal to 0 when an obstacle prevents light from reaching the surface unit area.

The integral of the light transport equation is expensive to compute. When real-time rendering is aimed, a crude approximation can be made to give the illusion of environment lighting with soft self-shadowing of objects: ambient occlusion. This approximation assumes that incident radiance is constant for every direction,  $L_i(P, \omega_i) = L$ , and BRDF is constant (lambertian surface,  $f_r(P, \omega_o, \omega_i) = \rho_d/\pi$ , see Section 2.2.1). The light transport equation

then becomes:

$$L_o(P, \omega_o) = L_e(P, \omega_o) + L \frac{\rho_d}{\pi} \int_{\Omega_+(N)} V(P, \omega_i) \cos \theta_i d\omega_i \quad (2.30)$$

Ambient occlusion is defined as follows:

$$AO(P) = \frac{1}{\pi} \int_{\Omega_+(N)} V(P, \omega_i) \cos \theta_i d\omega_i \quad (2.31)$$

$AO(P) = 1$  when the visibility is equal to 1 for every incidence direction. The light transport equation using constant radiance of the environment, lambertian surface and ambient occlusion becomes:

$$L_o(P, \omega_o) = L_e(P, \omega_o) + L \cdot \rho_d \cdot AO(P) \quad (2.32)$$

This equation is very inexpensive to compute when rendering, hence leading to its adoption in the movie industry and by video games. Ambient occlusion can be stored per vertex and interpolated per fragment. It can also be stored in texels of a texture that is mapped on the lit object. An alternate storage scheme for ambient occlusion is presented by Kontkanen and Laine [KL05] on their work on ambient occlusion fields that store occlusion data in a volumetric field around lit objects.



## 2.1.5 Global illumination

The previous sections considered surface lighting only due to light sources with or without occlusion. Such computations are called direct lighting. However, surfaces in any scene are illuminated not only by light sources, but also by light bouncing on neighbor surfaces. The contribution of neighbor surfaces is important for realism in rendered scenes, but is very computationally expensive: if  $N$  surface unit areas are present in a scene ( $N$  is large),  $N(N - 1)$  interactions between surfaces have to be considered for a single bounce of light.

As mentioned earlier, in the absence of any participating medium in the scene, radiance along a ray is constant. This means that the outgoing radiance at a point of a surface in a given direction contributes to the irradiance of a point of another surface. For a given surface of the scene, all other surfaces are then considered as light sources. The outgoing radiance from these surfaces is actually due to the incoming radiance from the light sources and the other surfaces. This shows that the relation is recursive.

The light transport equation (Equation 2.11) can be written a different way:

$$L = L_e + \mathcal{T}.L \tag{2.33}$$

where  $L$  is the equilibrium light in the scene,  $L_e$  is the light due to emission of the light sources and  $\mathcal{T}$  is the light transport operator. Solving Equation 2.33 for  $L$  gives:

$$L = L_e(1 - \mathcal{T})^{-1} \tag{2.34}$$

Expansion of this equation to a Neuman series gives:

$$L = L_e + \mathcal{T}.L_e + \mathcal{T}^2.L_e + \mathcal{T}^3.L_e + \dots \quad (2.35)$$

$\mathcal{T}.L_e = L_{direct}$  represents the first bounce of light after leaving the light sources. The previous equation can be rewritten in term of direct lighting:

$$L = L_e + L_{direct} + (\mathcal{T}.L_{direct} + \mathcal{T}^2.L_{direct} + \dots) \quad (2.36)$$

The content of the parenthesis represents the second and higher order bounces of light, it is considered as  $L_{indirect}$ . This part is usually the most expensive term to evaluate. In most of the scenes, the evaluation of the first bounce of indirect light is enough ( $\mathcal{T}.L_{direct}$ ), the next bounces only affect the result slightly.

## 2.2 Material properties

Each surface in an environment has its own material properties affecting incident light scattering and reflection. In this section, we present the notion of BRDF as well as several BRDF models used for different types of materials. We then talk about BTFs for describing materials of spatially varying properties.

### 2.2.1 BRDFs

Bidirectional Reflectance Distribution Functions (BRDFs)  $f_r(P, \omega_o, \omega_i)$  give a description of reflected light distribution at a point  $P$  of a surface given an incident light flux direction  $\omega_i$  and a reflection direction  $\omega_o$  (Figure 2.4). These 4D functions depend on the local micro-geometry. A surface material defined by a BRDF does not vary spatially, the BRDF parameters are the same everywhere on the surface. The next section describes representations of spatially varying materials.

BRDFs have two important properties: reciprocity and energy conservation. The reciprocity rule (Helmholtz reciprocity) gives:

$$f_r(P, \omega_o, \omega_i) = f_r(P, \omega_i, \omega_o) \quad (2.37)$$

for all pairs of directions  $\omega_o$  and  $\omega_i$  in  $\Omega_+(N)$ . The energy conservation rule requires that the total energy of reflected light is less than or equal to the energy of incident light. For every direction  $\omega_o$  in  $\Omega_+(N)$ :

$$\int_{\Omega_+(N)} f_r(P, \omega_o, \omega_i) \cos \theta_i \, d\omega_i \leq 1 \quad (2.38)$$

Reflectance (also known as albedo) is defined as follows:

$$\rho(\Omega_i, \Omega_o) = \frac{\int_{\Omega_o} \int_{\Omega_i} f_r(\omega_o, \omega_i) \cos \theta_o \cos \theta_i \, d\omega_i d\omega_o}{\int_{\Omega_i} \cos \theta_i \, d\omega_i} \quad (2.39)$$

where  $\Omega_i$  and  $\Omega_o$  can be a single direction, a solid angle or the whole hemisphere.

If  $\Omega_i$  is the whole hemisphere and  $\Omega_o$  a single direction, hemispherical-directional reflectance is obtained and gives the total reflection in direction  $\omega_o$  due to constant illumination. It is a 2D function defined as follows:

$$\rho_{hd}(\omega_o) = \frac{1}{\pi} \int_{\Omega_+(N)} f_r(P, \omega_o, \omega_i) \cos \theta_i d\omega_i \quad (2.40)$$

If  $\Omega_i$  and  $\Omega_o$  are the whole hemisphere, the hemispherical-hemispherical reflectance is obtained and is a single constant value that represents the fraction of incident light reflected by a surface when the incident light is constant from every direction. This value is defined as:

$$\rho_{hh} = \frac{1}{\pi} \int_{\Omega_+(N)} \int_{\Omega_+(N)} f_r(P, \omega_o, \omega_i) \cos \theta_o \cos \theta_i d\omega_o d\omega_i \leq 1 \quad (2.41)$$

A BRDF is called isotropic when only one azimuthal angle is necessary ( $\phi_i$ ,  $\phi_o$ , or the difference between these two angles).

Several BRDF models have been designed, used for different types of materials and for different purposes. Some of them are defined ad hoc, making them easy for a designer to configure, others use a more physically-based approach.

## Lambertian surfaces

Lambertian surfaces are perfect diffuse surfaces that scatter incident light uniformly in every direction. This model is well-adapted to matte surfaces. It is also a very simple

analytical model that simplifies many calculations related to the light transport equation.

The BRDF of a lambertian surface is defined as:

$$f_r(P, \omega_o, \omega_i) = \frac{\rho_d}{\pi} = k_d \quad (2.42)$$

where  $\rho_d \leq 1$  is the diffuse albedo of the surface (hemispherical-hemispherical reflectance in this case).  $k_d$  is the diffuse BRDF and  $k_d \leq 1/\pi$ . The hemispherical-directional reflectance can be analytically determined:

$$\rho_{hd}(P, \omega_o) = \frac{\rho_d}{\pi} = k_d \quad (2.43)$$

### Phong reflectance model

The Phong's reflectance model [Pho75] is an isotropic ad hoc model, using simple and intuitive parameters to be defined, which is adapted to plastic-like shiny materials. It is the sum of a diffuse reflection term (as for lambertian surfaces) and a specular term. Since the Phong's model does not respect the energy conservation rule, it has been modified [Lew94] and is then defined as follows:

$$f_r(P, \omega_o, \omega_i) = k_d + k_s (\omega_r \cdot \omega_o)^s \quad (2.44)$$

$$= \frac{\rho_d}{\pi} + \frac{\rho_s(s+2)}{2\pi} (\omega_r \cdot \omega_o)^s \quad (2.45)$$

where  $\rho_s$  is the specular albedo (with  $\rho_d + \rho_s \leq 1$ ),  $k_s$  the specular reflectance coefficient,  $s$  is the shininess giving an information about the size of the specular highlights (the higher, the smaller are the highlights), and  $\omega_r$  is the ideal reflection vector defined as follows:

$$\omega_r = 2(\omega_i \cdot N)N - \omega_i \quad (2.46)$$

### Blinn-Phong reflectance model

The Blinn-Phong model [Bli77] is a variation of the Phong reflectance model. Rather than using the dot product between the ideal reflection direction  $\omega_r$  and the reflection direction  $\omega_o$ , the dot product between the surface normal  $N$  and the halfway vector  $H$  is used. The  $H$  vector is halfway between  $\omega_i$  and  $\omega_o$ . It can be computed as follows:

$$H = \frac{\omega_i + \omega_o}{\|\omega_i + \omega_o\|} \quad (2.47)$$

The reflectance model is then defined as:

$$f_r(P, \omega_o, \omega_i) = k_d + k_s (N \cdot H)^s \quad (2.48)$$

The shininess  $s$  in this expression is different from the Phong's model since the halfway angle is smaller than the angle used by Phong. The value  $s$  can be scaled to achieve similar results between the two models. This model is the one used in the fixed pipeline of 3D graphics cards, through interfaces as OpenGL and DirectX.

## Microfacet models

The microfacet models make the assumption that some rough surfaces can be modeled a large set of microfacets. Microfacets are considered very small compared to  $dA$ . Therefore, the aggregate behavior of the facets determines the scattering. The most important parameters are the statistical distribution of the facets and the reflection model of the individual facets.

Torrance and Sparrow [TS67] consider that each facet is a perfect mirror and use the Fresnel's law  $F_r(\omega_o)$ . The Torrance-Sparrow model is adapted to metals and brushed metals and is defined as follows:

$$f_r(P, \omega_o, \omega_i) = \frac{D(H)G(\omega_o, \omega_i)F_r(\omega_o)}{4 \cos \theta_o \cos \theta_i} \quad (2.49)$$

where  $D(H)$  gives the probability that a microfacet has its normal equal to  $H$ ,  $G(\omega_o, \omega_i)$  is the geometric term taking self-shadowing into account.  $F_r(\omega_o)$  can be approximated using the Schlick's formula [Sch94]:

$$F_r(\omega_o) = f_0 + (1 - f_0)(1 - (\omega_o \cdot N))^5 \quad (2.50)$$

where  $f_0$  is the Fresnel factor at normal incidence.

The Oren-Nayar model [ON94] considers that microfacets are lambertian surfaces and that their orientation distribution is based on a Gaussian with  $\sigma$  as the standard deviation

of the orientation angle. The BRDF model is defined as follows:

$$\begin{aligned}
 f_r(P, \omega_o, \omega_i) &= \frac{\rho}{\pi} (A + B \max(0, \cos(\phi_i - \phi_o)) \sin \alpha \tan \beta) \\
 A &= 1 - 0.5 \frac{\sigma^2}{(\sigma^2 + 0.33)} \\
 B &= \frac{0.45\sigma^2}{\sigma^2 + 0.09} \\
 \alpha &= \max(\theta_i, \theta_o) \\
 \beta &= \min(\theta_i, \theta_o)
 \end{aligned} \tag{2.51}$$

## 2.2.2 SBRDFs and BTFs

Many materials are not uniform; the reflection properties change for each point of the surface. If surfaces are flat enough, Spatial Bidirectional Reflectance Distribution Functions (SBRDF) can be used. They use a BRDF model for each point of a surface, which creates a 6D function. McAllister et al.[MLH02] use the Lafortune BRDF model [LFT97] for each BRDF. This representation enables compact storage of a SBRDF using textures for hardware-accelerated rendering since each lobe of the Lafortune model uses only a direction vector and specular exponent. McAllister [McA04] explains how to implement the rendering step efficiently using fragment shaders that compute the local BRDF for each fragment.

Bidirectional Texture Functions (BTF) are adapted to spatially varying surfaces with meso-structures, which are larger than micro-structures modeled by BRDFs and smaller



than macro-structures modeled using geometry. The BTF can be considered as a SBRDF where the BRDF for each point of the surface is multiplied by a visibility function to take into account self-occlusion. The BTF is then a set of Apparent BRDFs (ABRDF).

BTFs are 6D functions that require large storage space. The visibility functions introduce high frequencies that make the data larger to keep a good accuracy. Many storage and compression schemes exist for BTFs, presented in the survey of Müller et al. [MMS04]. An intuitive storage scheme is a set of images, one for each reflection and incident light direction. By performing interpolation of the images for arbitrary reflection and incident light directions, an approximation of the surface reflectance can be obtained. Sets of ABRDFs can be stored using BRDF models adapted to measured data or using compression, using PCA (Principal Component Analysis) for example.

## 2.3 Volume rendering

For many years, 2D textures have often been used to represent reflection parameters of surfaces. For flat surfaces, this representation is correct. However, the interaction of light with fur for example 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. To render fur on objects like a teddy bear, Kajiya and Kay [KK89] introduce volumetric textures. They use a reference volume that is repeated over an underlying surface made of bilinear patches. The rendering method they propose is raytracing. When a ray

is cast and encounters an instance of the reference volume, the ray 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 value cumulated along the ray. An area with density  $\rho$  (portion of space occupied by geometry) crossed on a length  $L$  has a transparency  $e^{-\tau\rho L}$  where  $\tau$  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 main drawback of the raytracing method is the high rendering time it requires if dedicated hardware is not used. By discretizing the optical equation, using slices, classical hardware texturing capabilities can be employed. Slices aligned with the viewer are rendered with blending so accumulation can be performed. According to Ikits et al. [IKL04],

accumulated color and opacity ( $C$  and  $A$ ) can be computed as follows:

$$C = \sum_{i=1}^n C_i \prod_{j=1}^{i-1} (1 - A_j) \quad (2.52)$$

$$A = 1 - \prod_{j=1}^n (1 - A_j) \quad (2.53)$$

where  $C_i$  and  $A_i$  are the color and opacity of the  $i^{\text{th}}$  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  $i + 1$ . These expressions can be iteratively evaluated for a rendering of the samples from back to front (Over operator) using the following equations:

$$\hat{C}_i = C_i + (1 - A_i)\hat{C}_{i+1} \quad (2.54)$$

$$\hat{A}_i = A_i + (1 - A_i)\hat{A}_{i+1} \quad (2.55)$$

For front to back rendering (Under operator):

$$\hat{C}_i = (1 - \hat{A}_{i-1})C_i + \hat{C}_{i-1} \quad (2.56)$$

$$\hat{A}_i = (1 - \hat{A}_{i-1})A_i + \hat{A}_{i-1} \quad (2.57)$$

$\hat{C}_i$  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.

With recent hardware, the raytracing approach can be used in pixel shaders, in conjunction with 3D texturing. The book of Engel et al. [EHK06] detail many approaches for real-time volume rendering with lighting using these hardware features.

## CHAPTER 3: PREVIOUS WORK

In this chapter, we present works related to nature modeling and rendering, particularly related to our two main topics: grass and trees. We give advantages and drawbacks of the different methods, so we can show more accurately what are our contributions.

### 3.1 Nature modeling and rendering

Nature modeling and rendering has been studied for many years. Plant modeling studies individual plants with an emphasis on their structure. For example, L-systems [Lin68, PL90] use a grammar to describe geometrical structures and are well adapted to plants, even for the simulation of their growth. L-systems are now commonly used in commercial packages [XFr] to create models of flowers, trees, bushes, etc.

Plants are never alone in an environment, they have neighbors on terrains with which they have interactions. The full study of these interactions would give enough information to simulate full ecosystems and get the position and size of each individual plant. Deussen et al. [DHL98] have developed a software architecture to model and render such ecosystems using L-systems and interaction rules between plants. To distribute plants over the ground,

they coarsely represent them as circles that represent ecological neighborhoods. 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. Water presence under the ground influences the distribution of plants. Lane and Prusinkiewicz [LP02] improved this model and extended L-systems to handle populations of plants and trees.

Rendering full ecosystems is a heavy task due to the amount of geometry to process and the complexity of light interactions between each surface in the scene. Rendering of nature scenes has already been achieved using raytracing and a cluster of computers [DCD05, DMS06]. The main drawbacks of this method is the required computing power, which is excessive for the common user, and the presence of aliasing due to the geometric approach.

## 3.2 Grass rendering

Grass is one of the nature elements that is difficult to render in real-time due to its geometric complexity. Hundreds of millions of grass blades are present on any nature scene, park, golf course, sport stadium, etc. Brute force rendering is not advised due to the required computing power and the potential aliasing due to the small size of the grass blades. Since the processing power to account for every single blade is not yet available, many approximations have been used to accelerate rendering, reducing at the same time the overall quality of the resulting images. We present hereafter some of the existing methods. Grass can be seen from different viewpoints: near the ground, at walker's height, from the sky, etc. That

means grass blades can have large projections on screen or can be less than a single pixel wide. The different types of algorithms we present are dedicated to specific ranges of view distances.

### 3.2.1 Geometry-based rendering methods

Geometry-based methods allow for precise representations of objects, animated if desired, with textures and lighting. However, the processing power required to render many thousands of grass blades in real-time is nowadays unavailable. Rendering lots of triangles for grass blades that 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 easier management of different grass species, with variable length and density.

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

Procedural modeling is well suited to geometric grass rendering. In this case, the grass designer has very few parameters to modify to get detailed surfaces of grass. Moreover, memory requirements are low. Particle systems [Ree83, RB85] consist of clouds of simple

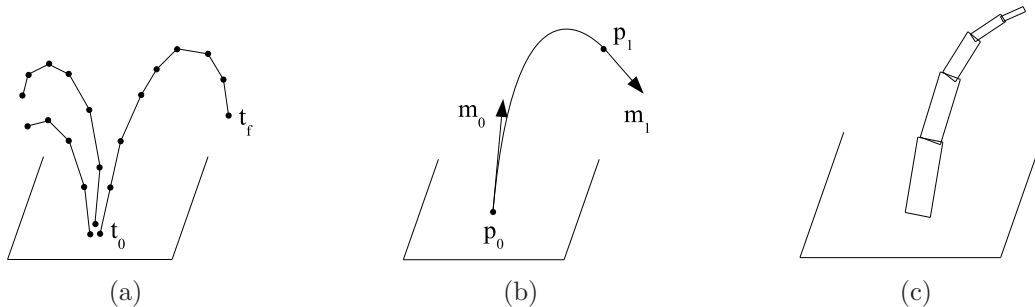


Figure 3.1: Grass blades using geometry. (a) Using trajectories of particles. The dots represent the generated coordinates. (b) Using a cubic Hermite curve. (c) Using a chain of billboards.

primitives that 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: particles have an initial speed toward a random direction when created by an emitter. The generated coordinates, to be assigned to the particles, can be used to describe the shape of grass blades: the initial time  $t_0$  represents the root of the blade, while the final time  $t_f$  the end. An example is shown in Figure 3.1(a).

Another possible method to determine the coordinates of the grass blade vertices is the use of cubic Hermite curves [AH02]. These curves are defined by the coordinates  $(p_0, p_1)$  and the tangents  $(m_0, m_1)$  of their ending points (Figure 3.1(b)). The coordinates of the points  $p(t)$  along the curve are, for  $t \in [0, 1]$ :

$$p(t) = (2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)m_0 + (t^3 - t^2)m_1 + (-2t^3 + 3t^2)p_1 \quad (3.1)$$

$$p(0) = p_0, \quad p(1) = p_1, \quad \frac{\partial p(t)}{\partial t}(0) = m_0, \quad \frac{\partial p(t)}{\partial t}(1) = m_1 \quad (3.2)$$



Several rendering methods exist for grass blades. When using particles systems, Reeves and Blau [RB85] propose 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 [DCS02] for the rendering of plants constituted of thin and long structures. These structure constraints are applicable to grass blades. Color variations of these anti-aliased lines can simulate lighting. At the top of the grass blades, ambient and diffuse components are at their maximum. When going down to the ground, these components decrease.

Using triangles for each grass blade is also possible. This approach is commonly used in the movie industry since there is no real-time constraint for the rendering. Lighting is easy in this case since any lighting algorithm dedicated to geometry can be used. Real-time raytracing using a cluster of computers [DCD05] is also a possibility but requires expensive equipment and presents important aliasing.

Another method makes use of colored billboards. A chain of five billboards allow for efficient rendering of a grass blade [PC01, GPR03] (Figure 3.1(c)).

The dynamic behavior of grass is mainly due to the wind and other events such as crushing by walkers or vehicles, and can be simulated with procedural approximations or physically-based models. Procedural approaches are generally used for real-time cases. For example, Pelzer [Pel04] uses trigonometric functions to approximate the swaying of grass blades. Perbet and Cani [PC01] 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. Due to the visual complexity of grass, exact physically-based animation is very time-consuming.

### 3.2.2 Image-based rendering methods

Grouping several grass blades and rendering them as a single primitive is a way to increase the rendering speed. Image-based approaches are particularly adapted to this task. A texture contains a set of grass blades that can be rendered on a single polygon using texturing. However, this approach is adapted only for rendering of far objects due to the visible flatness and upsampling artifacts due to the image insufficient resolution when covering an important part of the rendering window. The grass blades in the image can be created from photographs or procedurally generated.

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 by Perbet and Cani in [PC01], the drawback of this method is that lighting and animation are not realistic.

Billboard is the main structure that has been used for a long time, it is a single or a couple of quadrilaterals covered by a semi-transparent texture that represents a whole plant or a part of it. The rendering of a whole plant is then reduced to the rendering of a couple of textured primitives. Semi-transparency allows the representation of non-rectangular objects. The lighting model used with these billboards has been simple in most of the approaches,

but nowadays per-pixel computations appear and allow more precise lighting computations, even if still approximative.

Simple billboards are used in many old video games due to their low rendering cost. A single quadrilateral is rendered with a semi-transparent texture which is always facing the camera [Wha05]). A constraint is often added to keep the represented object vertical. When two billboards are close, popping artifacts appear when the camera is moving due to their intersections, therefore making simple billboards adapted to sparse objects only. Moreover, this method shows no parallax when the camera is moving, decreasing the realism of the rendered images.

To show more parallax, several quadrilaterals can be rendered. This kind of configuration is retained by Pelzer in [Pel04]. He uses three vertical quadrilaterals creating a star when 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.

Aligned layers of billboards are simpler and consist in placing textured layers of vertical strips in the same orientation [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 reduce pattern repetition, this method introduces many artifacts, in particular if grass is viewed from above.

Rendering quadrilaterals covered with 2D textures is a very fast operation. Therefore, more complex lighting models can be used without drastically affecting interactivity too much.

By adding an alpha channel to BTF data, this reflectance function can be used for billboards, rather than simple semi-transparent textures. Meyer et al. [MNP01] use a set of images per view and light direction to represent the BTF data. This approach requires interpolation of several images at rendering time, and is limited by the low sampling of the BTF. Shah et al. [SKP05] also make use of BTFs to render grass but use a compression scheme to get a higher sampling rate of the BTF. They define only a single layer of grass parallel to the ground, which can be considered as a texture map augmented with reflectance information. The BTF data are compressed using PCA (Principal Component Analysis). Their grass representation also contains depth information to handle silhouettes of grassy terrains and occlusions with external objects.

As image-based rendering uses static images, animation cannot be performed in real-time applications. For animation purpose, new images have to be computed which could be very time-consuming. In case of simple texture mapping of grass over the terrain, geometry of the underlying terrain can be deformed to simulate the animation effect. In fact, terrain is not really deformed but the light reflection parameters are dynamically modified.

To animate simple and cross billboards, their associated quadrilaterals have to be bent. Pelzer [Pel04] proposes to move the upper part of cross billboards (called grass objects). The lower part should not be moved otherwise roots of grass blades would move and look

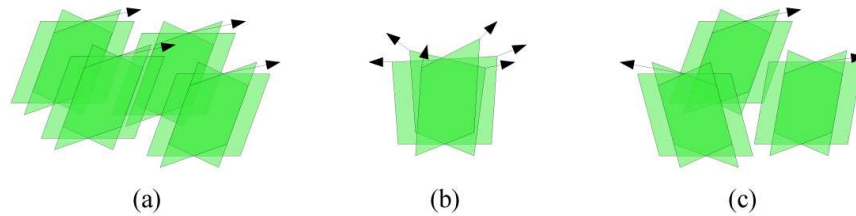


Figure 3.2: Three kinds of animation of grass objects from [Pel04]. (a) Per cluster of grass objects. (b) Per vertex. (c) Per grass object.

unnatural. Pelzer proposes three kinds of deformation. The first one consists in animating clusters of grass objects (Figure 3.2(a)). Every top vertex of a cluster is 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 3.2(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 3.2(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.

Aligned layers of billboards can be deformed for animation [PC01]. Not only the top of the billboards is animated (Figure 3.3(a)) but also the middle levels using polygon strips (Figure 3.3(b)). This method is efficient for distant grass viewed at walker's height.

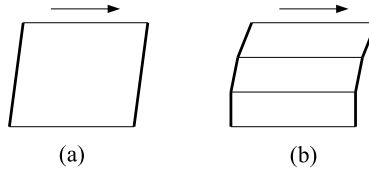


Figure 3.3: Two kinds of billboard deformation. (a) Quadrilateral billboard with shifted top. (b) Strip of polygons with progressive deformation.

### 3.2.3 Volume-based rendering methods

Methods based on volume images offer more parallax effect than billboard approaches. A base surface is subdivided into a bilinear patch, a 2D grid inside which each square contains an instance of a reference volume. This latter is called *texcell* (to avoid confusion with *texel* that is a term employed in other papers [KK89, Ney96] and that originally represents a texture element). Despite the fact that a large amount of data is required to represent grass accurately, volume rendering can be used for grass at middle distance from the viewer since it usually offers pleasant results with few aliasing. Volume rendering handles very efficiently complex objects with many repetitive details, hence is adapted to grass rendering. Volume representations offer full parallax: when the viewer is moving, objects are correctly rendered with no flatness as with billboards.

Lengyel et al. [LPF01] use series of semi-transparent textured shells to give the illusion of volumetric rendering for fur. These textures are created with slicing of the original geometry using clipping planes. Rather than using silhouette texcells [DN04], they use a fin texture, a texture containing a lateral view and used for the silhouette of objects.

Bakay et al. [BLH02] also use a stack of shells. However, their approach is less memory-intensive: they use a single texture to render ground and blades of grass with different lengths. Quadrilaterals with the same texture are rendered several times with alpha test enabled and different alpha thresholds depending on the height. The texture is used entirely at ground level. For the upper shells, only the positions of the gray dots in alpha channel correspond to rendered texels.

Fillrate is an important bottleneck for the two previous methods. In spite of the relatively low number of rendered pixels due to the alpha test, shells rendering time is high because every quadrilateral is fully rasterized. Moreover, these methods do not handle lighting easily.

Another representation is the set of slices presented by Meyer and Neyret [MN98] and partially used by Decaudin and Neyret [DN04] to render forest landscapes. A set of 2D textures stands for a volume. An advantage is the possibility to render such slices with every 3D card contrary to 3D textures that need modern hardware.

To convert a patch of grass to volume data, slicing with the standard hardware-accelerated rendering pipeline can be performed [MN98]. 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 of the slice.

Volume rendering is a computationally expensive 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 [DN04] to render trees on hills but the position of the sun cannot change.

Our goal is to render illuminated grass. Consequently, we will study some of the existing volume illumination methods. As proposed by Neyret [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. Kajiya and Kay’s reflectance model [KK89] is represented by a thin cylinder, well suited for fur. The local orientation is equal to the fur’s hair direction. Neyret [Ney95b] uses ellipsoids to represent a more general reflectance model. With them, different reflectance function shapes can be modeled, from spheres to approximated cylinders. Furthermore, 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 Normal Distribution Functions (NDFs) [Ney98]. They represent the distribution of normals to the surfaces inside a voxel.

The previous lighting techniques are originally dedicated to raytracing. Variants of these algorithms can be used for real-time applications. By using slices of the reference volume, 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 [IKL04, KPH03]. The slicing axis is set halfway between the view and light directions. For each rendered slice, two passes are performed. The first pass renders 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.

Space-deformations of a reference volume (texcell) mapped onto a surface create a thick skin of complex objects. Neyret [Ney95a] introduces simple methods to animate these texcells: deformation of the underlying surface, animation of height vectors allowing for deformation of the whole volume space, successive steps of a simple motion using a set of reference volumes.

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 empirical functions. Usually, functions inspired by real grass behavior are defined and 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.

Meyer and Neyret [MN98] use slicing to render grass at interactive rates and apply these animation techniques, particularly the second one.

### 3.3 Tree rendering

Tree rendering has been studied due to its use in many applications. Several levels of realism have been achieved when rendering, depending on the targeted use. We present hereafter some of the research work related to tree rendering and lighting.

#### 3.3.1 Geometry-based modeling and rendering methods

Description of a tree using only geometry is rarely used due to the impact on rendering performance. Many thousands of leaves are present in a common tree, and branches also need to be modeled. This approach is used when rendering speed is not important, typically in the movie industry where quality of the resulting images is important.

Modeling of trees can be done manually but becomes tedious as soon as leaves are concerned. That is why procedural generation is often used. L-systems are widely used to model several types of trees (presented in Section 3.1). Weber and Penn [WP95] propose an intermediate approach, where trees are generated with an ad hoc model configured with a set of user-defined parameters. These parameters are chosen to be user-friendly and allow procedural generation of the tree geometry rather than defining each polygon individually. However, these parameters remain tedious to manipulate when not knowing exactly the meaning of each of them.

Okabe et al. [OOI05] present a very user-friendly approach where a graphical user interface replaces parameter editing. First, the user draws a 2D sketch of the tree. Then, an algorithm creates a 3D model using the sketch and some rules, such as maximizing the distance between branches. Finally, the user adds, cuts and removes branches by a succession of mouse clicks directly on the rendered 3D model. This approach is not based on biology but allows quick modeling according to the wish of the user.

Galbraith et al. [GMW04] use a completely different approach. Rather than modeling trees directly with polygons, they use implicit surfaces. Their approach is well-suited for animation of growing trees and accounts for some singularities such as branch bark ridges. However, this method is not suited to real-time rendering.

Wang et al. [WWD05] use geometric modeling for their high-quality rendering of plant leaves, directly applicable to trees. They use a PRT approach [SKS02] for high quality shading in real-time. However, this method is expensive for large trees and forests of trees and requires a long preprocessing.

Scanned images are often used as textures for the branches and the leaves, they help getting results as photo-realistic as possible.

### 3.3.2 Image-based rendering methods

Methods based on images are more common in interactive applications. Depending on the distance from the viewer, two common representations are used: leaves as billboards or trees as billboards.

#### Leaves as billboards

In this representation, each leaf is represented as a simple primitive, often a quadrilateral, which is textured using a semi-transparent image. The trunk is often modeled with regular geometry to avoid a flat appearance. This method is very close to the geometric representation, except that the semi-transparency of the leaf textures decreases the number of processed polygons. Additional speed-up can be achieved by grouping leaves together in a single image.

Habel et al. [HKW07] use this approach to achieve real-time rendering using a physically based model for the translucency of the leaves. They use a new representation of the leaf material parameters to achieve fast evaluation of subsurface scattering.

This approach is also the one we use to represent high quality trees. Each leaf or group of leaves are represented as billboards and lighting computations are performed per billboard.

#### Trees as billboards

This representation uses a single billboard or a set of crossed billboards to represent a whole tree (as presented in Section 3.2.2). This approach is useful for trees that are far from

the viewer, since leaves projected to the screen are often less than one pixel wide. Moreover, this rendering method is very fast. However, these billboard schemes present some flatness due to the projection of billboard images on flat polygons. This lack of parallax effect has been addressed in several research works.

To increase this parallax effect, Jakulin [Jak00] propose to use a set of orthogonal slices, similar to methods of volume rendering based on slicing. The transparency of the slices is changed to keep most of the billboards orthogonal to the view direction as opaque as possible. Slices parallel to the view direction are faded out to hide the potentially visible flatness. The trunk and the branches are rendered using regular geometry because flatness would be perceived too much using the image-based rendering approach. Borse and McAllister [BM02] improved the algorithm, particularly the blending part, to be usable with stereographic systems.

Max and Ohsaki [MO95, Max96] use a hierarchy of multi-layer Z-buffers to reconstruct an approximation of the original tree geometry using billboards augmented with multiple depth images. The advantage of this method is the better parallax effect compared to regular billboards due to the depth information. They also embed coverage masks to achieve anti-aliasing. Rather than storing the whole tree as billboards, they use a hierarchical representation where the lowest level corresponds to single leaves, and next levels represent set of branches of increasing size.

Qin et al. [QNT03] use billboards augmented with many additional buffers containing depth, normals, shadow masks, etc. to reconstruct the tree. This approach gives very in-

interesting results, with handling of many parameters related to lighting: specular reflections, transmission, inter-reflections of leaves and shadows. However, even if this approach accelerates rendering compared to traditional geometry, rendering ten thousand trees requires several minutes.

### 3.3.3 Volume-based rendering methods

Decaudin and Neyret [DN04] use volumes to render trees over hills. They pre-render trees inside a 3D texture that is tiled to cover hills. Decaudin and Neyret [DN04] use two kinds of texcells (instances of the 3D texture): regular ones and silhouette ones. 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 use slices parallel to the view plane. Vertex shaders are used to compute slices coordinates, hardware culling is used to display only necessary parts of the volume. Ikits et al. [IKL04] only employ the CPU to compute slices coordinates; it allows the exact computation of slice coordinates that do not go beyond the limits of the volume texture and it avoids the use of hardware culling. The main drawback of this rendering method is the application to dense forests only. Lighting is pre-computed in the 3D texture so no variation of lighting is possible. The number of used 3D textures is low, therefore some uniformity is visible.

Reche et al.[RMD04] create volume data for a tree based on a set of photographs. Each voxel of the bounding volume of the tree contains an opacity and color information. At

rendering time, billboards are rendered for each texel using blending, in back-to-front order, using the *Over* accumulation operator (see Section 2.3).

### 3.3.4 Tree lighting

Lighting of trees is always a difficult operation due to the geometric complexity and the lighting complexity: materials of leaves have complex reflectance properties due to transmission through their multi-layered structure.

BTFs are interesting for billboard rendering. Meyer et al. [MNP01] propose 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 BTF textures are composed of leaves or trees viewed from different points with various light directions. When rendering, the three nearest sampled view vectors are selected. The same action is applied to the light direction. A blending is performed with the nine selected images taken from the pre-computed textures. The blending coefficients are the barycentric coordinates relative to the view and light directions. Objects are considered as directly seen from the camera, that is why the reflection vector used by the BTF is replaced by the view vector.

For realistic looks, a tree should be rendered using global illumination. Light from the environment entering into the foliage is scattered inside the tree since inter-reflections between leaves occur. However, global illumination is a very expensive process.

Ambient occlusion, presented in Section 2.1.4, is a cheap alternative to global illumination. It gives convincing visual results, even if not really accurate. Reeves and Blau [RB85] propose a model of ambient occlusion where attenuation of light from the environment is based on the distance from the currently rendered leaf to the border of tree. The farther from the border, the higher the number of leaves hiding the environment. Hegeman et al. [HPA06] define a model that is more complex. The tree is modeled by a sphere, the ambient occlusion term is computed per leaf based on its normal. The plane of the current leaf cuts the tree bounding sphere into two parts. The volume of the spherical cap on the side the leaf normal is pointing to is used to get an approximative attenuation factor. The higher the volume of the spherical cap, the higher the number of leaves that are attenuating light from the environment. Even if this approach gives better results than the Reeves and Blau's approach, it still looks inaccurate for many types of trees, particularly the ones that do not have the shape of a sphere or an ellipsoid, or sets of them.

Another approach based on artistic principles uses the aggregation of leaf normals to smooth the overall distribution of leaf orientation. The goal is to increase the visibility of the tree outline due mainly to global illumination. Luft et al. [LBD07] define the envelope of the tree using implicit surfaces, each leaf is considered as a metaball. Once the envelope is precomputed, the distance of each leaf to the border of the tree is used to estimate the ambient occlusion. The normal of each leaf is replaced by the normal of the closest point on the implicit surface. Therefore, lighting the leaves becomes equivalent to lighting the implicit surface. Peterson and Lee [PL06] use a similar approach. They sample the tree



branches into a sparse set of points. Based on a scalar function that decreases with the distance from the sample points, the new leaf normals are computed using the gradients of this scalar function. These two approaches are empirical and do not really handle indirect lighting.

Indirect lighting can be estimated more accurately using offline radiosity-based solutions [CAB98, SSB03]. However, these methods are not adapted to real-time rendering.

Patmore [Pat93] proposes a model in which foliage is considered as a participating medium where particles represent leaves. This model estimates the attenuation of light inside a set of leaves rather than directly computing it. However, this model does not estimate indirect lighting.

# CHAPTER 4: RENDERING GRASS IN REAL-TIME WITH DYNAMIC LIGHTING

## 4.1 Introduction

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 an important element to be studied when rendering natural 3D scenes due to its abundance in nature.

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 [RB85, PC01, BLH02, Pel04, SKP05] but with coarse approximations, or render grass in high quality but offline [DHL98]. 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



Figure 4.1: Park scene rendered in real-time using our grass level of detail scheme.

quality or a better speed. Arbitrary shaped surfaces of grass can be easily created using our management of grass density.

We start by explaining our algorithm, step by step. Then, we introduce some implementation issues and how we solve them. Finally, we present some rendering results followed by a conclusion and future work.

## 4.2 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 level of detail scheme (*LOD*) to achieve this goal. We start by presenting our global level 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. Then, we present our shadowing algorithm. Finally, we describe the handling of curved terrains.

### 4.2.1 Levels of detail

We want to render large terrains covered with grass such as football fields and golf terrains. Direct rendering of such huge amount of grass blade geometry in real-time using current hardware is impossible (about 4 minutes per frame for a grass field made up of 250 million grass blades), so we need to make extensive use of levels of detail. Also, we cannot define all parameters of every blade of grass of a terrain because the required memory size would be excessive. Therefore, we render multiple instances of a small *grass patch* over the whole ground surface [KK89, Ney96], laying on the cells of a uniform grid. To handle levels of detail, 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 4.2). The latter approach offers a good parallax effect: as seen from any direction, the grass patches do not look flat.

We use the distance from the camera as a criterion to switch between levels (Figure 4.2). For grass close to the camera, high quality grass blades are rendered using lit and shadowed geometry. When grass is farther, a large number of grass blades occupy small screen areas

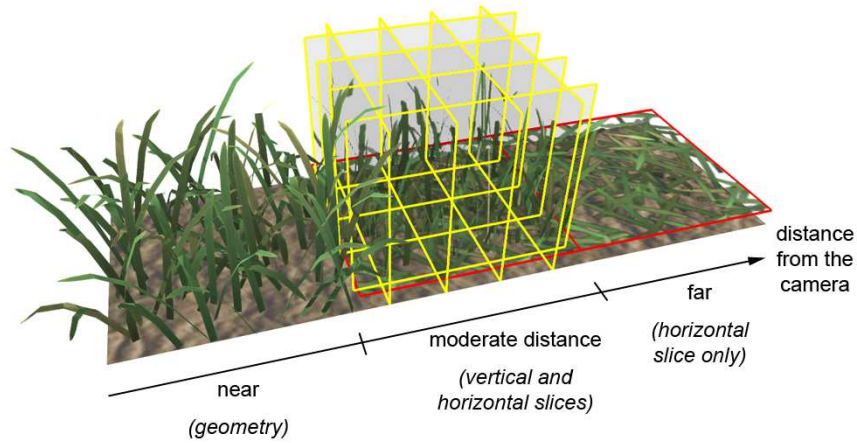


Figure 4.2: The three levels of detail, chosen depending on the distance from the camera. For nearby grass, simple geometry is used. At moderate distance, horizontal and vertical semi-transparent slices are rendered. For faraway grass, only the horizontal slice is used.

and rendering blade geometry becomes expensive. Hence, we resort to a volume rendering method using semi-transparent axis-aligned slices. For very far grass, the number of slices to be rendered becomes excessive. At such distances, grass surfaces have a flat textured appearance and hence we reduce the rendering cost by rendering only grass slices parallel to the ground.

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 4.2.3.

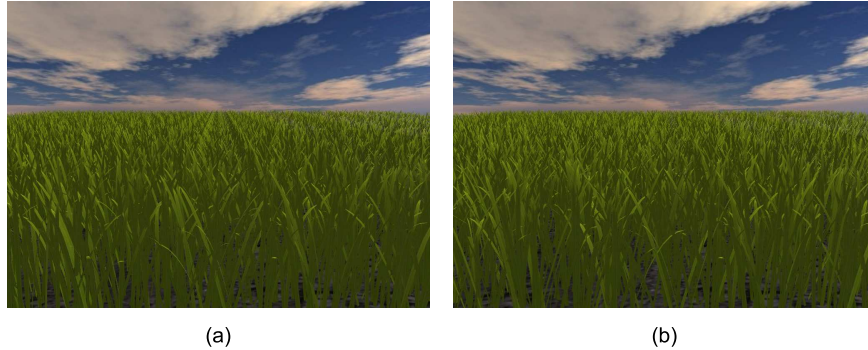


Figure 4.3: Result of aperiodic tiling. (a) Periodic tiling of a single grass patch. Notice the repetitive pattern at the center of the image. (b) Random mirroring of the grass patch instances to achieve aperiodic tiling. Repetitive pattern is no more noticeable.

A difficulty of any LOD scheme lies in the seamless transition management. We provide smooth transitions between the levels of detail using a density management scheme. This approach is described in Section 4.2.5.

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. One of the four versions is randomly chosen for each terrain grid cell (Figure 4.3). We define the four patches as mirrored versions of the base patch, so data are present only once in memory, the mirroring operation is done at run-time on the GPU. Such random orientations break the strong repetitive patterns. 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 at no memory overhead and almost no computational overhead. We chose this approach instead of Wang

tiling [CSH03] because Wang tiling requires rendering of several versions of the base patch. Multiple base patches mean multiple geometry and multiple volume representations. This significantly increases our memory overhead.

If the distribution of the grass blades is not uniform inside the grass patch, the dense clumps of grass appear in a regular fashion over the terrain, even when using randomly mirrored patches. So 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 (that represent 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 grid cell.

Some parameters can be defined globally over the terrain. For example, we use a *color map*, with texture repetition if desired, to modulate the color of the grass. As in Figure 4.26, a color map can be used to simulate painted grass on a football field. The material color of the grass blades is multiplied by the color taken from this map. For non painted grass, we use a color map containing a slight noise to break the regularity of the grass color over the terrain (used in Figure 4.25).

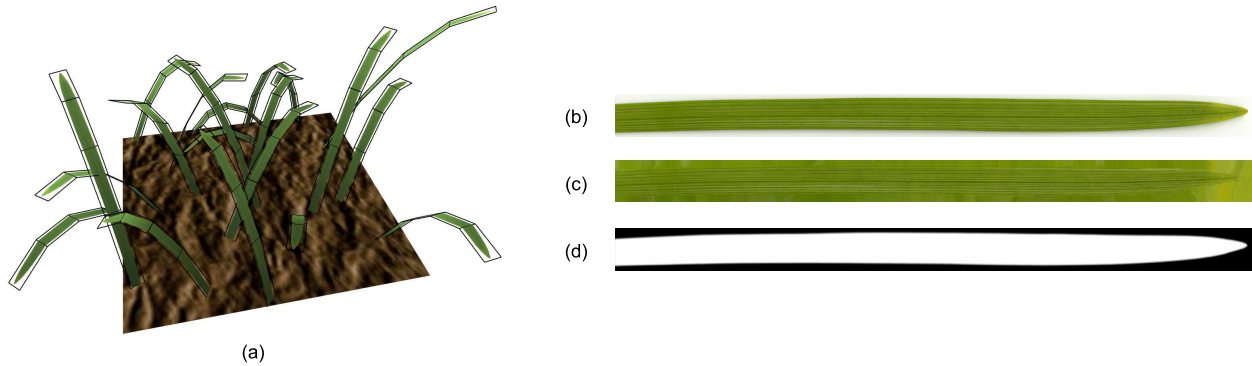


Figure 4.4: (a) Grass blades defined with textured semi-transparent quadrilateral strips. (b) Original scanned grass blade. (c) Color channel of the blade texture modified to remove the white border. (d) Alpha channel of the texture defining the shape of the blade.

### 4.2.2 Geometry-based rendering

Geometrically modeled grass blades are used for rendering close to the camera and for generating volume slices data. We model a grass blade by a *quadrilateral strip* as shown in Figure 4.4(a). Real grass blades are very thin: we approximate them with two-sided quadrilaterals of zero thickness. The trajectory of a particle system [Ree83, RB85] defines the shape of the strip: a particle is launched from the root of the blade, almost vertically, under the influence of 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 4.4(d)) of the texture covering the quadrilateral strips (Figure 4.4(c)) gives the correct shape of a grass blade. In Section 4.3.2, we detail our method to perform order-independent rendering of the semi-transparent quadrilaterals used for the grass blades.



The Lambert reflectance 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 projection in camera space. The color of each blade is slightly modified to simulate different ages and levels of degradation.

We add a component to this reflection model to approximate light transmittance through the grass blades. We consider the grass blades as very thin and of uniform thickness. We add to the reflected radiance at a point of a grass blade a term that is a function of the normal of the blade back face; we also scale the blade diffuse reflectance to account for the color change of the light traversing the grass blade fibers. The last term of Equation 4.1 describes this additional component in more detail.

For a single omnidirectional light source applied to a point of the surface of a grass blade, the resulting radiance  $I_r$  is:

$$\begin{aligned}
I_r &= I_{ambient} + I_{diffuse} + I_{transmitted} \\
&= K_d A_O I_a + K_d \max(N \cdot L, 0) \frac{I_d}{1 + \beta d^2} + K_d \gamma \max(-N \cdot L, 0) \frac{I_d}{1 + \beta d^2} \quad (4.1) \\
&= K_d A_O I_a + K_d \left( \max(N \cdot L, 0) + \gamma \max(-N \cdot L, 0) \right) \frac{I_d}{1 + \beta d^2}
\end{aligned}$$

where  $K_d$  is the diffuse reflectance factor of the grass blade material,  $A_O$  the ambient occlusion factor at the current point of the grass blade,  $I_a$  the intensity of the ambient light,  $N$  the normal to the grass blade surface,  $L$  the light direction from the surface point,  $\gamma$  the factor that is multiplied by  $K_d$  to account for the color change of light traversing the grass blade fibers,  $I_d$  the intensity of the point light source,  $d$  the distance between the surface point and the light source, and  $\beta$  the light attenuation factor. The 1 in the denominator avoids an infinite intensity when the lit point is very close to the light source.

To simulate ambient occlusion  $A_O$  along grass blades, we set the ambient occlusion factor to a low value close to the ground [RB85] 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 and then is interpolated per fragment.

### 4.2.3 Volume rendering

We use volume rendering for grass at intermediate distance from the camera, where rendering of individual grass blades is too expensive because of their high 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 thousand blades of grass. The width and depth of the volume 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 4.2.1.

Generally, methods using hardware 3D acceleration resort to slicing, where several slices of the volume are rendered using a 3D texture. A classical approach is to make the slices facing the camera [IKL04]. These slices are made semi-transparent to allow the visibility of the slices behind and to define the global shape of the objects inside the volume. With our instancing approach, many patches at different locations on the terrain have to be rendered. Consequently, every coordinate would have to be computed for each slice of every visible instance of the volume. This approach requires too much computation; additionally, linearly interpolated 3D texture accesses in hardware are expensive.

Therefore we use 2D slices aligned with the three axes (middle of Figure 4.2). 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.

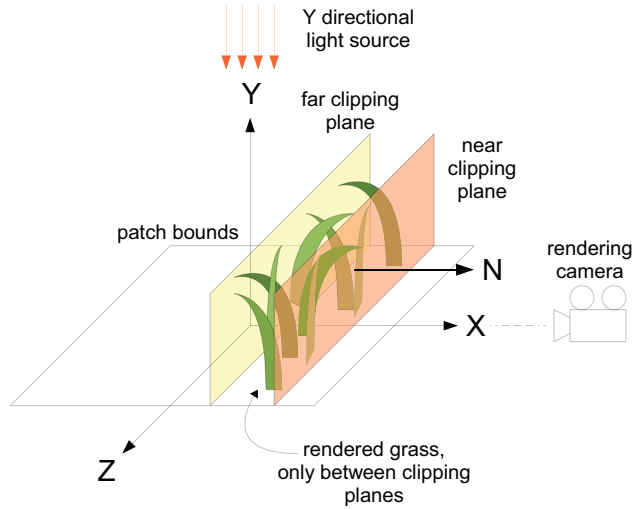


Figure 4.5: Rendering of a slice orthogonal to the  $X$  axis. The current orthographic camera is in front of the slice ( $X+$ ). The current light direction is  $Y$ . The normal to the slice is  $N$  and determines the front face of the slice. The near and far clipping planes allow the rendering of 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 volume representation does not store any vertex information of the grass blades. We define the slice textures using semi-transparent BTFs rather than simple 2D images as for classical billboards. BTFs are originally dedicated to the representation of macro-structures on a surface, so they can represent small variations of height with simulation of self-shadowing and self-occlusions. BTF is a 6-dimensional function defining the occlusion modulated 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

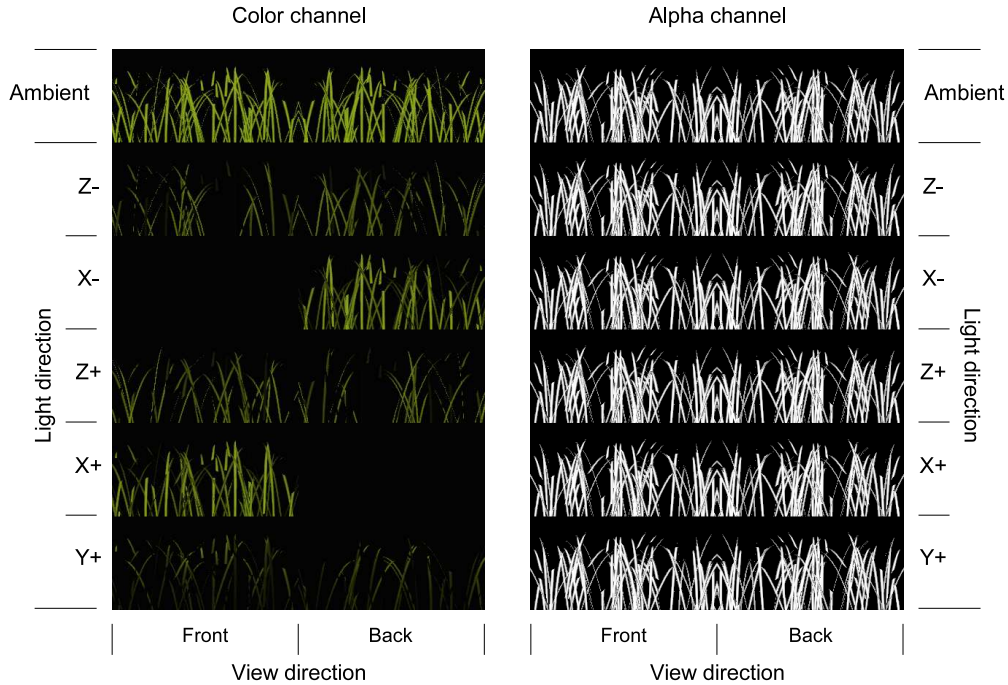


Figure 4.6: BTF data for a slice orthogonal to the  $X$  axis. The left image is the color channel, the right one is the alpha channel of the texture that stores a representation of the BTF. The five light directions are shown along the vertical axis, the view directions along the horizontal axis. Front view of a slice corresponds to the camera being on 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.

of zero thickness, we use only 2 view directions from 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 4.5). An example data set for a slice orthogonal to the  $X$  axis is shown in Figure 4.6. Even though the sampling is very low (five directions), the results using these slices are visually pleasing for grass surfaces with diffuse only reflectance.

To create the slices data, we resort to a method similar to that of Meyer and Neyret [MN98] 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 is done only once and the resulting slices data are stored on the hard drive. The method for a slice orthogonal to the  $X$  axis is illustrated in Figure 4.5. Light direction is one of the BTF parameters. Thus we use a directional light source to illuminate the grass blades. For the same reason, an orthographic camera is used rather than 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 4.5) but using geometry with a constant white color on a black background and using the alpha channel of the blade texture (Figure 4.4(*d*)). We obtain black and white images (right in Figure 4.6) with thin gray gradients at the grass blade borders. The gradients allow anti-aliasing when performing the final rendering (see Section 4.3.2). 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 4.6), 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 grass with slices, Equation 4.1 cannot be applied directly for lighting computation. Only data for 5 light directions are available in the BTF data so we need to interpolate to get the inbetween 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). The following equation is a modified version of Equation 4.1. This equation computes the diffused and transmitted radiance by combining BTF samples corresponding to each light direction  $L_i$ ,  $i \in \{1..5\}$ .

$$\begin{aligned}
 I_r &= K_d A_O I_a + \left[ \sum_{i=1}^5 \alpha_i K_d \left( \max(N \cdot L_i, 0) + \gamma \max(-N \cdot L_i, 0) \right) \right] \frac{I_d}{1 + \beta d^2} \\
 &= K_d A_O I_a + \left( \sum_{i=1}^5 \alpha_i C_i \right) \frac{I_d}{1 + \beta d^2}
 \end{aligned} \tag{4.2}$$

The  $K_d$  term is taken from the first row of the BTF image (Figure 4.6). This term is also used for the final ambient component computation. The  $K_d (\max(N \cdot L_i, 0) + \gamma \max(-N \cdot L_i, 0))$  term is taken from the five last rows of the image (called  $C_i$  in Appendix A) and is used for the diffuse component computation.  $\alpha_i$  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 only once for the whole grass surface. If the lighting computation due to a point light source is performed per pixel, computation of the previous equation is very expensive. Per vertex computation of the  $\alpha_i$  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.

Interpolation of BTF samples depending on the view direction is not useful since only two directions are defined (representing 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 4.6). Otherwise, the back face is rendered (right column of Figure 4.6).

For grass that is very far from the camera, particularly for viewpoints at high altitude, we use a restricted version of our volume rendering approach: only the horizontal slice is rendered for each visible patch. This approach looks similar to classical 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.

#### 4.2.4 Management of non-uniform distribution of grass

In nature, grass is never uniformly distributed over the ground. Various external phenomena introduce chaos: type of ground, availability of water [DHL98], rocks, roads, etc., hence



affecting the grass *density*. We define grass density as the number of grass blades per unit of surface. The density of grass at each point on the ground, which we call *local density*, can be defined with a *density map* (left of Figure 4.7).

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

A user defined density map (left of Figure 4.7) gives sufficient information to create arbitrary distributions of grass (right of Figure 4.7) on the terrain. This map is equivalent to a probability distribution function. We use bilinear interpolation to get the local density for each point on the ground. A simple way to simulate spatially varying grass density would be to change the opacity of uniformly distributed 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 blades distribution globally uniform.

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. To be able to do this, we introduce the notion of *density threshold* (Figure

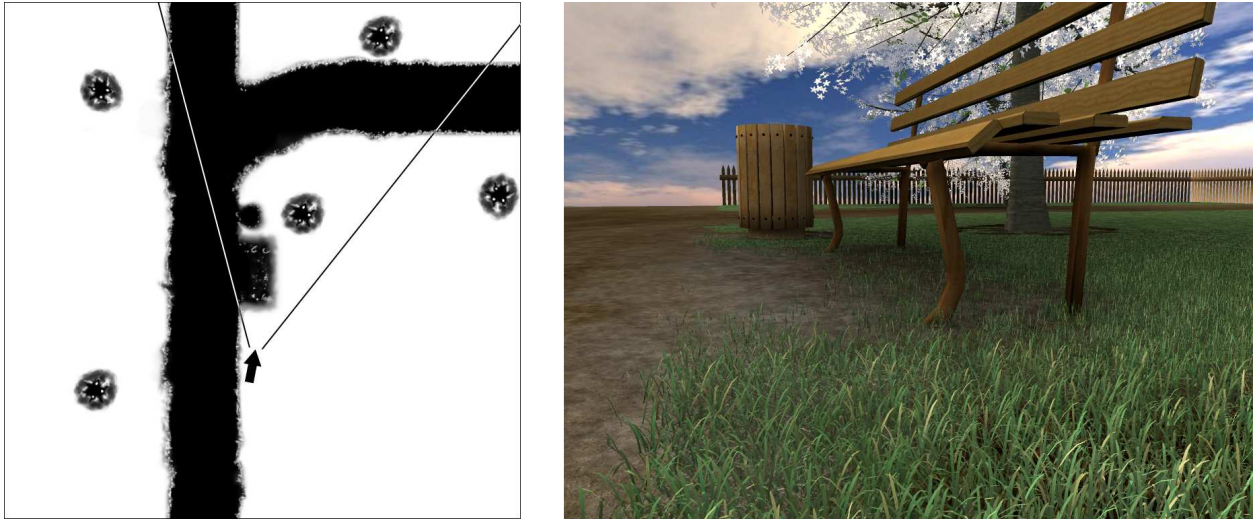


Figure 4.7: Modeling of grass using a bilinearly interpolated density map. The left image is an example of density map to render the scene of the right image. The arrow represents the camera position. Black pixels represent regions with no grass, white pixels represent the regions with maximum density.

4.8(a)). With each blade of the base grass patch we associate a threshold value ranging in  $]0, 1]$ . During the rendering step, for each blade of each rendered patch, we perform a test before rasterization: if the blade density threshold is strictly greater than the local density taken from the density map, then the blade is eliminated (Figure 4.8(b)). The blades with high threshold are then rendered only in locations with high local density. To keep the uniformity of grass distribution for any density value, we define the density thresholds randomly using a uniform distribution inside the single patch. To manage different species of grass, we use a density map for each of the species and render the final scene in multiple passes.

Handling density for grass rendered with geometry per blade will incur very high CPU overhead (comparisons with the local density and draw calls for each visible blade). So we

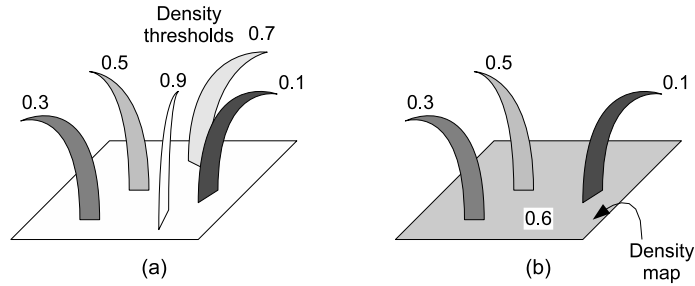


Figure 4.8: Density threshold management. (a) Each grass blade is assigned a density threshold in  $]0, 1]$ . (b) Rendering of this patch of grass using a density map with a constant value of 0.6. Only blades with threshold less than or equal to 0.6 are rendered.

handle density using the GPU. This requires a single draw call for each grass patch. The density threshold is stored in each vertex of a grass blade, with a constant value along the blade. The bottleneck of grass rendering with geometry is vertex processing (space transformations, lighting, shadowing), so we move the comparison between the density threshold and the local density to the fragment shader. If the comparison fails, the fragment is clipped. Additionally, local density is read from a texture, which is more efficient in a fragment shader than in a vertex shader.

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 4.9(b)). Texels on this texture covered by a grass blade are assigned a density threshold in  $]0, 1]$ , stored as gray levels. Every texel belonging to the same grass blade 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 of a pixel, we compare the value from the density threshold channel

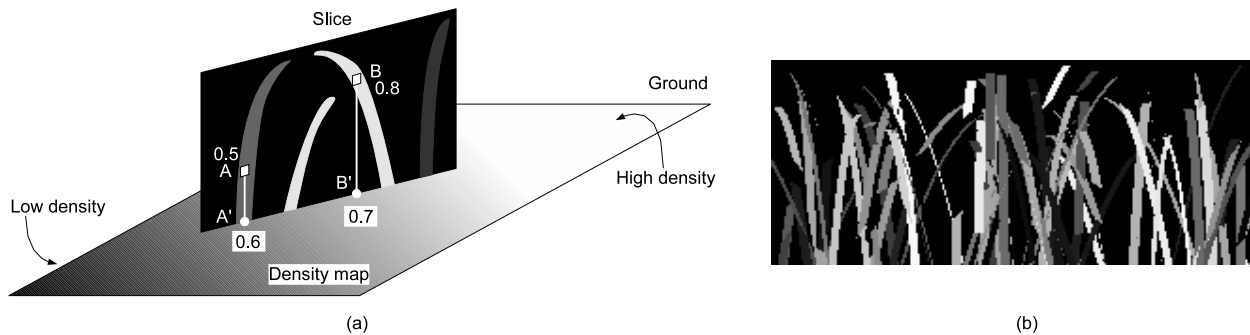


Figure 4.9: Handling of density threshold for the rendering of slices. (a)  $A$  and  $B$  are the pixels to be tested. Their projections onto the ground are  $A'$  and  $B'$ . With the given density values,  $A$  is displayed, but  $B$  is not. (b) Example of data for a slice.

(Figure 4.9(b)) with the local density read from the density map. If this density threshold is strictly greater than the local density, we discard the fragment. In Figure 4.9(a), 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, whereas the pixel  $A$  is displayed because its density threshold (0.2) is less than the local density (0.3).

#### 4.2.5 Seamless transition between levels of detail

Our level of detail scheme combines both geometry-based and volume-based rendering to render large grass terrains in real-time. Binary choice between one method and another makes the transition visible. So, smooth transitions between the levels of detail are desirable (Figure 4.10). A simple approach will be to fade from a rendering method to the other one by progressively changing the opacity of the grass blades depending on the distance from

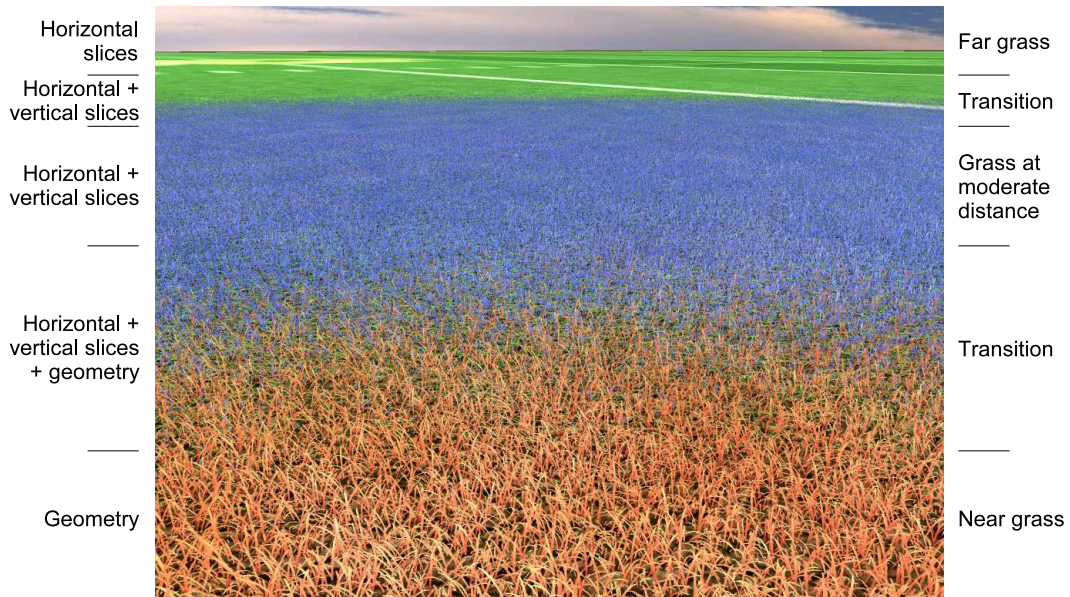


Figure 4.10: Smooth transitions between levels of detail made visible with false colors. Grass blades rendered with geometry are red. Vertical slices are blue. Horizontal slices are green.

the viewer. However, the results do not look natural due to the presence of semi-transparent grass blades.

We propose to extend and adapt our density management scheme, described in Section 4.2.4, to perform seamless transitions. In the region of transition between two levels of detail, the two corresponding rendering methods are used simultaneously. The local density for each grass blade is modulated by a weight function that depends on the distance from the blade to the viewer (Figure 4.11) and is different for each rendering method. For the first region of transition, a subset of the grass blades is rendered with geometry and the remaining blades are rendered with volume slices. In the second region of transition, vertical slices are progressively faded out since the opacity of the horizontal slices is increasing with

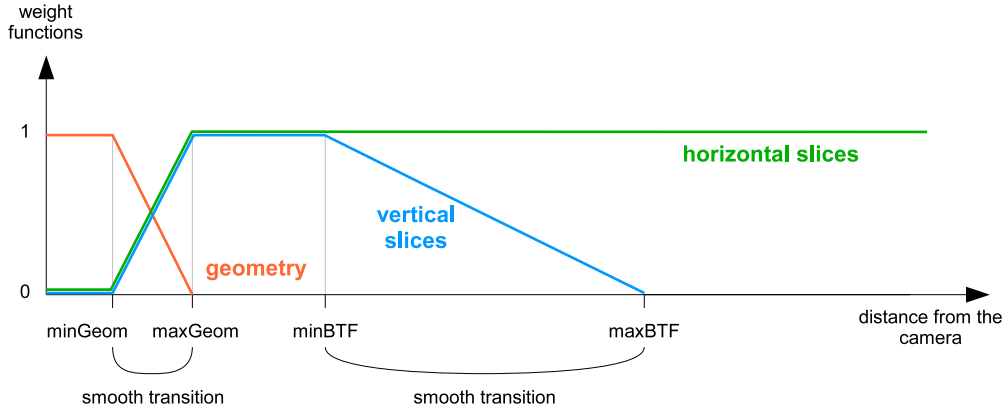


Figure 4.11: Functions weighting the local density for each rendering method, as a function of the distance to the viewer.  $minGeom$ ,  $maxGeom$ ,  $minBTF$  and  $maxBTF$  are user-defined parameters.

the distance from the viewer. This increased opacity compensates the missing vertical slices. This change of opacity occurs due to our custom mipmap filter described in Section 4.3.3.

The weight functions we use are defined hereafter. The parameters  $minGeom$ ,  $maxGeom$ ,  $minBTF$ ,  $maxBTF$  (see Figure 4.11) are defined by the user, depending on the desired rendering quality. The higher these parameters, the higher the rendering quality but the higher the rendering cost. We define  $d$  as the distance from the current grass blade to the camera,  $w_g(d)$  the weight function for grass defined by geometry. We define the weight function for geometry using the  $clamp(x, min, max)$  function as follows:

$$w_g(d) = clamp\left(\frac{d - maxGeom}{minGeom - maxGeom}, 0, 1\right) \quad (4.3)$$

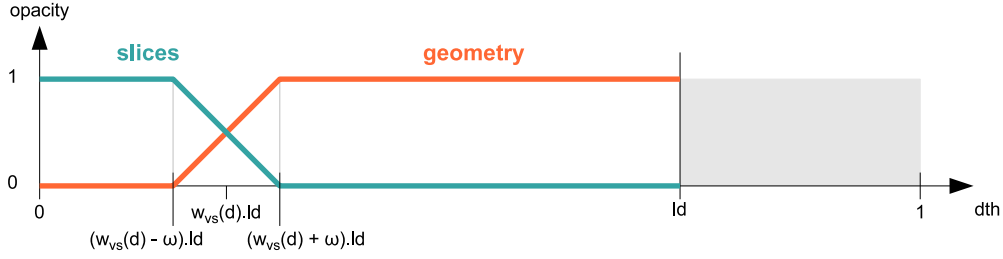


Figure 4.12: Distribution of grass blade opacities for each rendering method, as a function of the grass blade density threshold  $dth$ . If the density threshold of a blade is greater than the local density  $ld$ , it is eliminated (gray region).

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

$$w_{vs}(d) = clamp\left(\frac{d - minGeom}{maxGeom - minGeom}, 0, 1\right) \cdot clamp\left(\frac{d - maxBTF}{minBTF - maxBTF}, 0, 1\right) \quad (4.4)$$

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

$$w_{hs}(d) = clamp\left(\frac{d - minGeom}{maxGeom - minGeom}, 0, 1\right) \quad (4.5)$$

When moving the camera, some blades of grass previously rendered with geometry get rendered using slices, and vice versa. These sudden changes create popping artifacts. We address this problem by modifying the opacity of the grass blades around the region of transition between the two rendering methods. The opacity function has to keep the number of rendered semi-transparent blades as low as possible and to keep the local density management unchanged (we do not want semi-transparent grass blades due to local density handling). The function for the two rendering methods are defined in Figure 4.12.

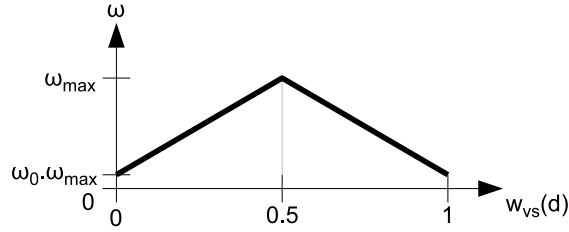


Figure 4.13: Values of  $\omega$ , the size of the transition region between rendering methods, as a function of the weight  $w_{vs}(d)$ .  $\omega_0$  is a user-defined offset to avoid a division by 0 when computing the opacity (Algorithm 1). We use  $\omega_0 = 0.05$  in our implementation.

Firstly, the grass blades for which the density threshold ( $dth$ ) is strictly greater than the local density ( $ld$ ) are eliminated. Among the remaining grass blades, the ones with  $dth$  less than  $w_{vs}(d).ld$  are rendered using slices and the others with geometry.  $w_{vs}(d) \in [0, 1]$  so we multiply it by the local density  $ld$  to get the transition point. We manage a window of width  $\omega$  around  $w_{vs}(d).ld$  to get a smooth transition between the levels of detail.  $\omega$  has to be small when the local density is small, thus we multiply it by  $ld$ . We change the opacity progressively in the range  $[(w_{vs}(d) - \omega).ld, (w_{vs}(d) + \omega).ld]$ .

For a fixed local density  $ld$ , if the weight  $w_{vs}(d)$  is close to 0 or to  $ld$ , the transition region goes out of range, creating persistent semi-transparent grass blades. We take care of this problem by decreasing  $\omega$  depending on its proximity to the bounds 0 and  $ld$ . The function for  $\omega$  is shown in Figure 4.13.

The final algorithm to handle density for horizontal and vertical slices is presented in Algorithm 1. These steps are executed at the beginning of the fragment shader, so it eliminates many expensive computations (lighting in particular) when the fragments do not have to be displayed. We consider  $w_{hs}(d) = w_g(d)$  for  $d \leq maxGeom$ .



---

**Algorithm 1:** Density management for slices rendering

---

```
if  $ld < 0.01$  then discard fragment;  
if  $dth > ld$  then discard fragment;  
 $\omega = \omega_{max} [\omega_0 + (1 - \omega_0) \cdot (1 - |2w_{vs}(d) - 1|)];$   
 $opacity = clamp\left(\frac{(w_{vs}(d) + \omega) \cdot ld - dth}{2\omega \cdot ld}, 0, 1\right);$   
if  $opacity < 0.01$  then discard fragment;
```

---

The algorithm for geometry rendering is the same except for the opacity. The following expression presents this difference (we consider  $w_{vs}(d) = 1 - w_g(d)$  for  $d \leq maxGeom$ ):

$$opacity = clamp\left(\frac{dth - (w_{vs}(d) - \omega) \cdot ld}{2\omega \cdot ld}, 0, 1\right) \quad (4.6)$$

## 4.2.6 Shadows

Shadows are an important part 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 relative 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 becomes prohibitive. We need to perform fast approximations that give visually pleasant dynamic shadows. There are three kinds of shadows: points on the ground occluded from light by grass blades (Figure 4.14), points of grass blades occluded from light by other blades (Figure 4.15) and self-shadowing of the blades. We do not manage



Figure 4.14: Shadows projected onto the ground.

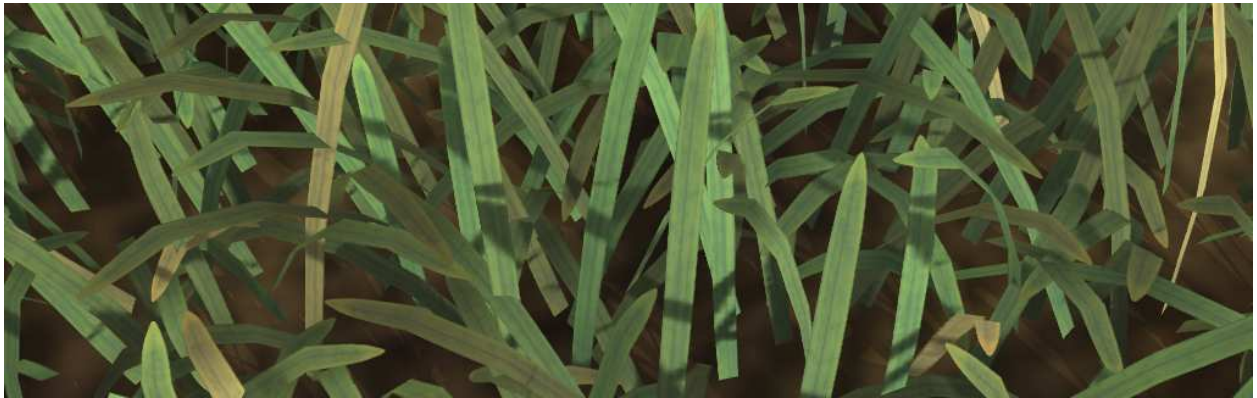


Figure 4.15: Shadows due to neighbor grass blades.

the latter because these shadows appear rarely due to the shape of grass blades. We use a different algorithm for each of the other kinds of shadows.

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. Firstly, 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 finally render a black quadrilateral with blending covering the screen to make the shadows visible. This method gives convincing results but only hard shadows can be rendered. For farther grass that uses volume slices, we use a similar method, but 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 since no soft shadows are handled, and will result in a uniformly shadowed ground.

The projective method cannot be applied to shadows cast by grass blades onto other blades because the shadow receivers are not planes. Algorithms dedicated to arbitrary surfaces may be used. For instance, we can use any shadow mapping technique [SD02] that uses a depth map from the light source viewpoint. However, it cannot be applied to grass since the required resolution of the shadow map texture is extremely high, otherwise strong aliasing artifacts will occur. The shadow volumes technique [Hei91] cannot be used for grass because the rasterization work is extremely time demanding for a high number of grass blades.

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 4.16). 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 4.16). The shadow mask is computed only once 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 thick slice of a grass patch defined by geometry is rendered between two clipping planes using an orthographic camera and a white background. This image is then mapped onto the cylinder at rendering time. Note that the rendered blades are assigned a black color.

At rendering time, for each vertex of a grass blade, a ray is traced from this vertex to the light source and the intersection point with the surrounding cylinder is computed. The coordinates of this point 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 4.15. Approximative soft shadows can be obtained at almost no cost: mipmapping is enabled for the shadow mask texture and a bias is applied to the mipmap level when accessing it.

The number of shadows received by a grass blade depends on the presence of neighbor blades. When the local density is low, the blade has almost no occlusions from light. We consider that the variation of local density around a grass blade is negligible compared to the radius of the shadow mask cylinder, so we define the following approximation: we modulate

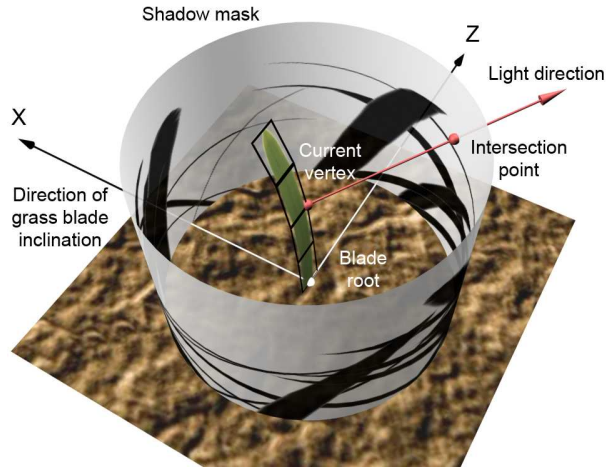


Figure 4.16: Shadow rendering step: projection of shadows coming from the neighbor grass blades. A shadow mask mapped onto a cylinder is used as a visibility function.

the value read from the shadow mask by the local density using the following expression:

$$finalShadowValue = 1 - (1 - valueFromShadowMask).localDensity \quad (4.7)$$

In case of volume rendering, shadows cast by blades onto other blades can be straightforwardly handled. In fact, the BTF images are generated using the shadow mask algorithm. As the volume rendering algorithm does not change, no additional cost is incurred.

A natural scene is never made of grass blades only. Additional elements in a scene, called *external elements*, can be additional light occluders for grass blades. To account for these occlusions, we make use of *ambient occlusion*. For a point on 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 on the ground, we provide

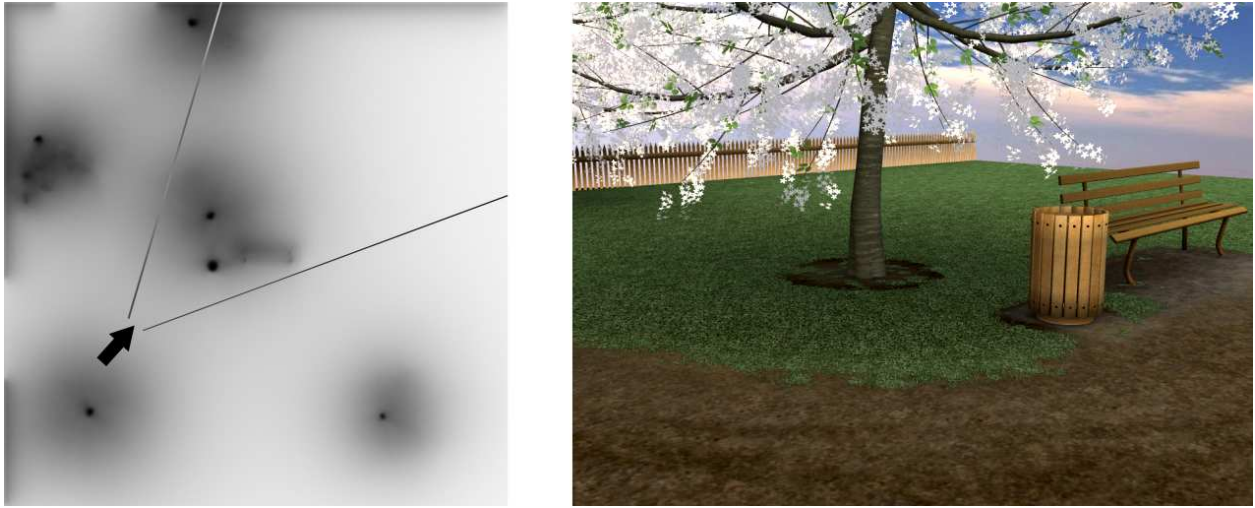


Figure 4.17: Ambient occlusion map covering the terrain and example of rendering. The arrow represents the camera position to get the image on the right.

the information through an *ambient occlusion map* covering the terrain (Figure 4.17). This map is computed only once in a preprocessing step since it is independent of the point light source direction (only the ambient light is considered). 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.

### 4.2.7 Management of the terrain

Creating multiple instances of a single grass patch over a terrain works well for a small flat surface. However, we come up with many issues when working with large terrains, particularly when they are not flat. We present these issues hereafter along with the solutions we propose.

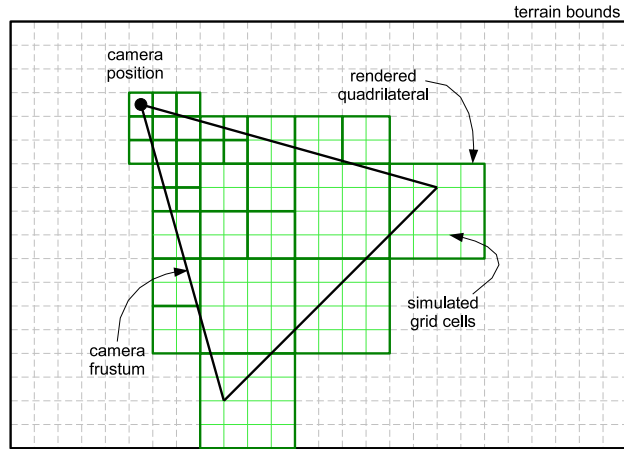


Figure 4.18: Terrain managed using a quadtree. The dark green quadrilaterals are rendered, determined by the camera position and orientation. The simulated grid cells are created using texture repetition.

#### 4.2.7.1 Macro-cells

For large terrains, the number of patches rendered for faraway grass is excessive. Additionally, brute force frustum culling of the uniform grid supporting the patches is too expensive. To reduce the culling and rendering times, we use a *quadtree* structure: a tree is built in a preprocessing step, each leaf contains the bounding sphere of a cell of the grid, each upper node contains the bounding sphere of a *macro-cell*, a  $2 \times 2$  group of children 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 4.18). 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 number of cells.

In section 4.2.1 we showed how aperiodic tiling is performed to mask repetition patterns over the terrain. Random mirroring for each grid cell is used, however it cannot be applied to macro-cells by simply repeating texture coordinates. So, we define a *patch orientation map* (Figure 4.19(a)), a texture mapped over the terrain where each texel corresponds to 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 mirroring is not used (Figure 4.19(b)), the coordinates  $(u', v')$  used to access the horizontal slice texture are defined as follows:

$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} \text{frac}(\text{patchSize} \times u) \\ \text{frac}(\text{patchSize} \times v) \end{pmatrix} \quad (4.8)$$

where  $\text{patchSize} = 4$  in Figure 4.19. To manage mirroring (Figure 4.19(c)), we propose the following modification of the previous equation:

$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} \text{frac}(\text{patchSize} \times \text{orientationMap}(u, v).\text{red} \times u) \\ \text{frac}(\text{patchSize} \times \text{orientationMap}(u, v).\text{green} \times v) \end{pmatrix} \quad (4.9)$$



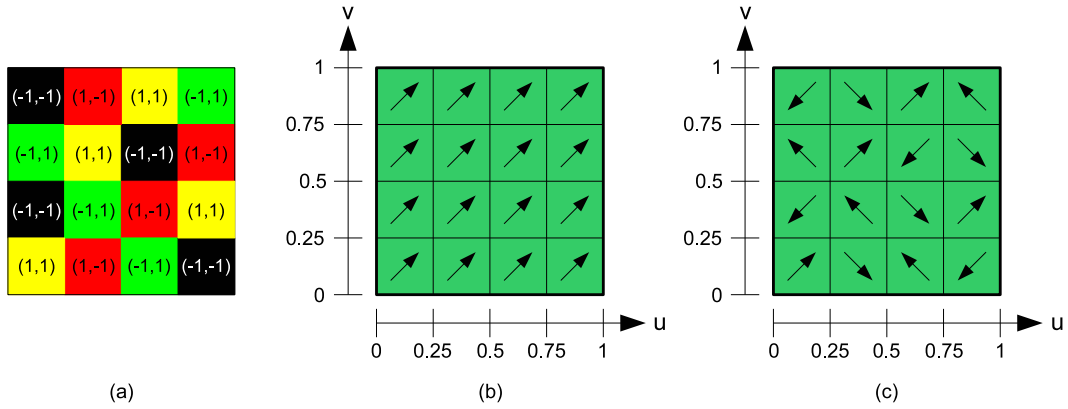


Figure 4.19: (a) Subset of a patch orientation map covering a  $4 \times 4$  cell. (b)  $4 \times 4$  macro-cell without mirroring. The repetition of the grass patches is visible. (c) Random mirroring applied to the grass patches, removing the repetition effect.

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  $1 \times 1$  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. Equation 4.9 is much simpler than the one for rotations.

#### 4.2.7.2 Distortion of the patches for curved terrains

For a flat terrain, only a translation in the plane of the terrain is necessary to place an instance of a grass patch in its corresponding cell. Thus, only a translation is necessary to transform a point from patch space to world space. As we do not want to be limited to flat grass terrains, we use a height map to represent the height of each grass patch corner. However, this introduces additional transformations due to the distortion of the patches (as

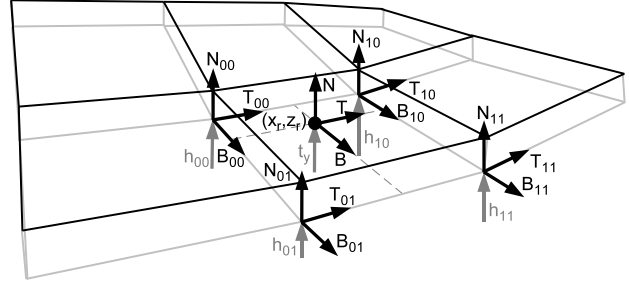
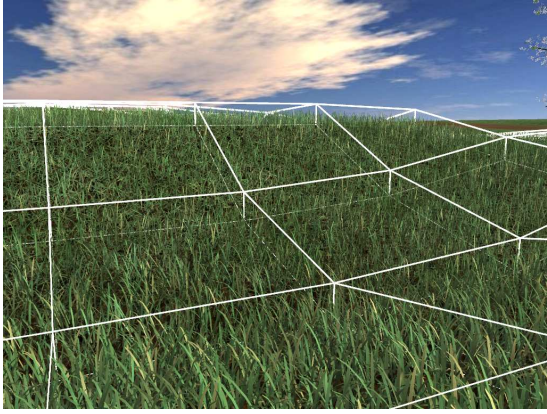


Figure 4.20: Distortion of the patches over the terrain. A coordinate frame  $(T, N, B)$  is defined for each point of the terrain, computed using the values at the current patch corners.

in Figure 4.20). Each point of the terrain is assigned a coordinate frame  $(T, N, B)$  (tangent, normal, binormal). Keeping the vector  $N$  vertical allows the grass blades to stay almost vertical even on high slope terrains (which is the case in nature). However the coordinate frame  $(T, N, B)$  does not remain orthogonal.

For each vertex of each grass blade of a patch of the terrain, an additional translation  $t_y$  along the vertical axis is applied (computed from the height map data). Each grass blade root has coordinates  $(x_r, 0, z_r)$  in the space of the patch, with  $x_r, z_r \in [0, 1[$ .  $(h_{00}, h_{01}, h_{10}, h_{11})$  are the heights for each corner of the current patch, given by the height map. The translation  $t_y$  is then computed through interpolation on a bilinear patch:

$$t_y = (1 - x_r).(1 - z_r).h_{00} + x_r.(1 - z_r).h_{10} + (1 - x_r).z_r.h_{01} + x_r.z_r.h_{11} \quad (4.10)$$

For lighting computations, the direction of the point light source from the current vertex is transformed from world space  $(l_{x_w}, l_{y_w}, l_{z_w})$  to patch space  $(l_{x_p}, l_{y_p}, l_{z_p})$  as follows:

$$\begin{aligned}
l_p = [T \ N \ B]^{-1} l_w &= \frac{1}{B_z T_x - B_x T_z} \begin{bmatrix} B_z & 0 & -B_x \\ B_y T_z - B_z T_y & T_x B_z - T_z B_x & T_y B_x - T_x B_y \\ -T_z & 0 & T_x \end{bmatrix} l_w \\
&= \frac{1}{B_z T_x - B_x T_z} \begin{bmatrix} (B_z, 0, -B_x) \\ B \times T \\ (-T_z, 0, T_x) \end{bmatrix} l_w \tag{4.11}
\end{aligned}$$

The  $[T \ N \ B]$  matrix cannot be inverted by simply taking the transpose since it is not orthogonal.

Once this equation is computed, the vector  $(l_{x_p}, l_{y_p}, l_{z_p})$  is normalized and used for lighting computations of the grass defined by geometry and to access the BTF data.

If we apply Equation 4.11 with a coordinate frame  $(T, N, B)$  constant along the patch, sudden variations of lighting appear between patches. Hence the interpolation of these vectors along the patch:

$$\left\{ \begin{array}{l} T = \text{normalize}((1 - x_r).(1 - z_r).T_{00} + x_r.(1 - z_r).T_{10} + (1 - x_r).z_r.T_{01} + x_r.z_r.T_{11}) \\ N = (0, 1, 0) \\ B = \text{normalize}((1 - x_r).(1 - z_r).B_{00} + x_r.(1 - z_r).B_{10} + (1 - x_r).z_r.B_{01} + x_r.z_r.B_{11}) \end{array} \right. \tag{4.12}$$

with  $(T_{00}, T_{10}, T_{01}, T_{11})$  and  $(B_{00}, B_{10}, B_{01}, B_{11})$  the tangents and the binormals at the current patch corners.

Handling non flat terrain requires a modification of the terrain quadtree. If it is managed as in Figure 4.18, view dependent popping artifacts appear along the hills due to the macro-cells that can be flat only. We augment the nodes of the quadtree with a flatness information. The cells of the terrain are considered flat if they lie in a plane. Then, a group of two by two cells is considered flat if the four cells are all considered flat and lie in the same plane (with a small threshold). When rendering, macro-cells are rendered only if they are considered flat, otherwise the quadtree is traversed deeper and smaller cells are rendered.

### 4.2.7.3 Terrain silhouettes

Our management of levels of detail presents an undesirable flatness at the top of hills when they are far from the viewer. This occurs because only horizontal slices are rendered, even if grass blades should be clearly visible. We handle this problem by rendering a couple of vertical slices at the hill's silhouettes (Figure 4.21). When traversing the terrain quadtree, if a patch is considered as belonging to the silhouette, additional vertical slices are rendered. Then, at the fragment shader level, a function depending on the angle between the view vector and the terrain normal is added to the weight function for vertical slices (Equation 4.4).

A point on the ground is considered as a part of the silhouette if

$$\begin{cases} |N \cdot \hat{V}| < \epsilon \\ \max BTF < d < \max Silhouettes \end{cases} \quad (4.13)$$

with  $N$  the normal of the terrain at the currently tested point,  $V$  the view vector,  $\hat{V}$  its normalized version,  $d = ||V||$ ,  $\epsilon$  the threshold below which vertical slices are displayed (0.2 in our case),  $\max BTF$  the parameter defining the starting point of the far grass zone (defined as in Figure 4.11), and  $\max Silhouettes$  a parameter that represents the distance for which grass blades are too small to be seen (less than one pixel high in screen space).

We modify Equation 4.4 to handle the hill's silhouettes:

$$\begin{cases} w_{distance}(d) = clamp\left(\frac{d - \min Geom}{\max Geom - \min Geom}, 0, 1\right) \cdot clamp\left(\frac{d - \max BTF}{\min BTF - \max BTF}, 0, 1\right) \\ w_{silhouettes}(d) = \left(1 - \min(|N \cdot \hat{V}|/\epsilon, 1)\right) \cdot H(d - \max BTF) \cdot H(\max Silhouettes - d) \\ w_{vs}(d) = \min(w_{distance}(d) + w_{silhouettes}(d), 1) \end{cases} \quad (4.14)$$

with  $H(x) = 1$  if  $x \geq 0$ ,  $H(x) = 0$  otherwise.

#### 4.2.7.4 Management of multiple types of grass

Managing multiple types of grass on the same terrain is straightforward: each type of grass is rendered over the same terrain but with different density maps (as in Figure 4.22). These

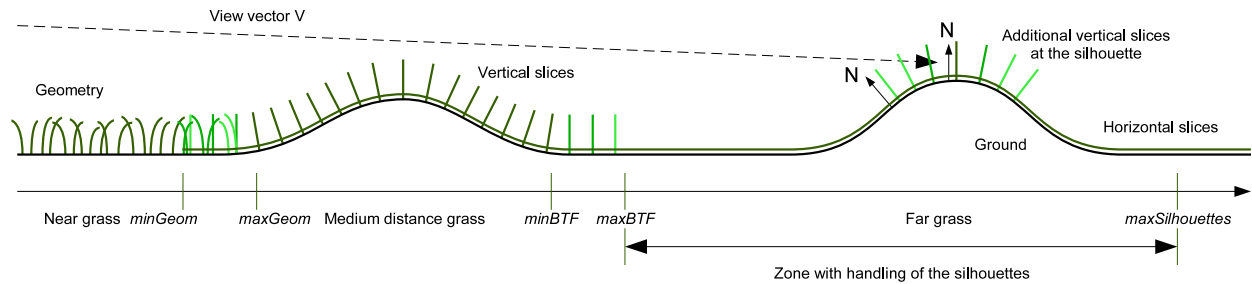


Figure 4.21: Vertical slices added at the top of hills. They appear depending on the angle between the terrain normal and the view vector and only for distances from the camera ranging from  $maxBTF$  to  $maxSilhouettes$ .

density maps can be painted by the user, which allows an easy modification of the distribution of each type of grass.

The height and density maps can also be automatically generated (Figure 4.22): we use a mesh for the terrain with material assignments for the faces, each material corresponding to a type of grass. The terrain mesh does not need to be a uniform grid. Rendering the terrain mesh from above using height as the pixel gray level gives the height map. To create the density maps, we render the terrain several times from above on a black background, with the faces of the current material rendered with a constant white color. Multisampling has to be enabled to allow smooth transitions from one type of grass to another.

## 4.2.8 Animation

In nature, grass often looks static, particularly when short. However, long grass blades can move under the influence of wind. To simulate the motion of grass in a patch we distort the

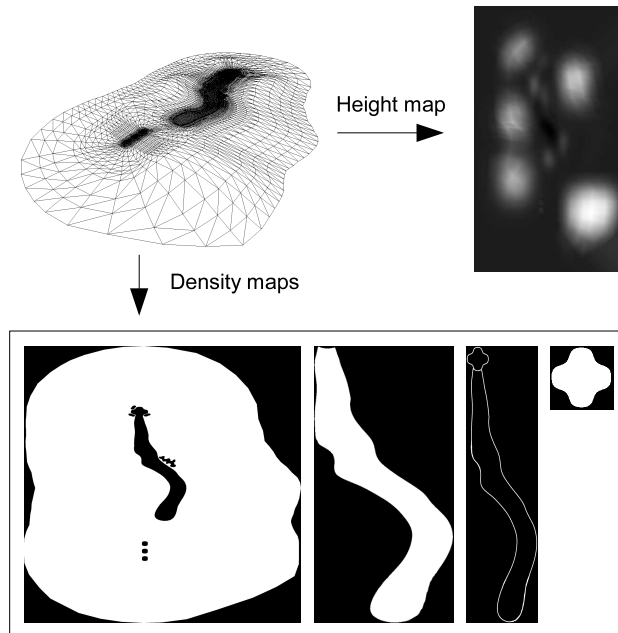


Figure 4.22: Generation of a height map and density maps from a mesh. For the height map, white represents the location of the highest points. Each density map is associated with a type of grass, white represents presence of grass.

patch using a method similar to the one presented by Meyer and Neyret [MN98] for volume rendering. It consists in shearing the base volume depending on a wind vector, as shown in Figure 4.23.

To define the wind vector for each volume corner, we can use a map defined over the terrain embedding the wind direction and strength for each point on the ground, as proposed by Perbet and Cani [PC01].

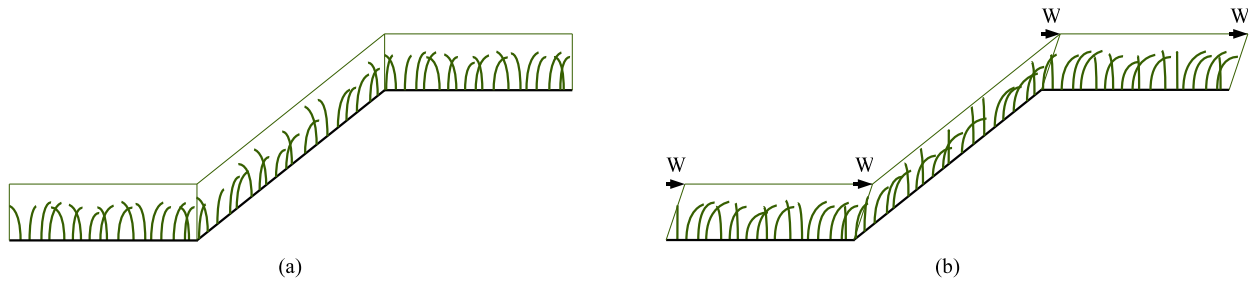


Figure 4.23: Animation of the grass patches. (a) Base position. (b) Distortions driven by wind vectors  $W$ .

### 4.3 Implementation

When implementing our grass rendering method, we encountered many issues, particularly related to filtering, mipmapping and aliasing. Speed issues were also of primary concern throughout. First, we describe our global grass rendering algorithm as well as the main shaders we use. Then, we present solutions to the filtering problems, enabling flicker free rendering of grass at any distance with adequate speed.

#### 4.3.1 Global grass rendering algorithm

This section gives an overview of our rendering algorithm, first describing the global steps of our approach then showing how vertices and fragments are processed for each level of detail.

The algorithm 2 presents the different initializations needed as well as the main rendering loop. At each frame, the terrain quadtree is processed to get the list of grass patches to render. Each patch of the list of visible patches is assigned a quadtree level, 0 corresponding to



the finest level (for every grass rendering methods), and a value greater than 0 corresponding to macro-cells (the size of a macro-cell is  $2^{\text{quadtreeLevel}} \times 2^{\text{quadtreeLevel}}$ ). The rendering loops for vertical and horizontal slices are written, as shown in Algorithm 2, to minimize the number of texture binding operations.

---

**Algorithm 2:** Initialization and main rendering loop

---

```

geomPatch = GenerateBladesGeometry(); // base patch generation (Section
4.2.2)
if BTF data already generated then LoadBTFData() else
GenerateBTFData(geomPatch);
if terrain already generated then LoadTerrainMesh() else
GenerateTerrainMesh(heightMap);
foreach rendered frame do
    pList = PatchesInsideCameraFrustum(currentCamera); // quadtree
    traversal (Section 4.2.7.1)
    RenderGround(pList);
    foreach patch  $p \in pList$  do
        if  $p.\text{quadtreeLevel} = 0$  then RenderPatchGeometry( $p$ );
    end
    foreach vertical slice  $s$  do
        Bind texture for slice  $s$ ;
        foreach patch  $p \in pList$  do
            if  $p.\text{quadtreeLevel} = 0$  then RenderPatchVerticalSlice( $p, s$ );
        end
    end
    Bind texture for horizontal slice;
    foreach patch  $p \in pList$  do RenderPatchHorizontalSlice( $p$ );
end

```

---

Algorithms 3 and 4 present the structures of the vertex and fragment shaders for the vertical slices. BTF interpolation coefficients are computed per vertex, interpolated and renormalized per fragment for efficiency reasons.

Shaders for the horizontal slices are similar, except for the following differences:

---

**Algorithm 3:** Vertex shader structure for the rendering of vertical slices

---

```
function verticalSlicesVertexShader()  
begin  
    Compute the vertex position in world and camera space (under influence of wind);  
    Compute the texture coordinates for the BTF texture, the density map and the  
    color map;  
    Get the light direction from the current vertex in patch space;  
    Determine the visible face of the slice (front or back);  
    Compute the BTF interpolation coefficients depending on the light direction in  
    patch space;  
end
```

---

---

**Algorithm 4:** Fragment shader structure for the rendering of vertical slices

---

```
function verticalSlicesFragmentShader()  
begin  
    Get the density threshold  $dth$  of the current grass blade and the local density  $ld$   
    on the terrain;  
    if  $ld < 0.01$  then discard fragment;  
    if  $dth > ld$  then discard fragment;  
    Compute  $w_{vs}(d)$  as in Equation 4.14;  
    Compute  $opacity$  as in Algorithm 1;  
    if  $opacity < 0.01$  then discard fragment;  
  
    Renormalize the BTF interpolation coefficients  $a_i$ ;  
    Apply to them the mirroring depending on the patch orientation map;  
    // Section 4.2.7.1  
    Apply the mirroring to the BTF texture coordinates depending on the patch  
    orientation map;  
    Get the BTF texture images  $BTF_i$  (front or back view);  
    Read values from the terrain color map, the terrain ambient occlusion map, the  
    BTF ambient component;  
     $fragmentColor = (sceneAmbient \cdot BTFambient \cdot terrainAmbientOcclusion$   
         $+ lightIntensity \cdot distanceAttenuation \cdot \sum_{i=1}^5 a_i \cdot BTF_i) \cdot localColor$   
    ; // Equation 4.2  
     $fragmentAlpha = BTFalpha \cdot opacity$  ;  
end
```

---

- the visible face of the slice is not determined, front face is always used
- texture coordinates account for the size of the macro-cells and for the orientation map (Equation 4.9)
- density for the silhouettes is not handled

For the rendering of grass blades using geometry, some computations that were performed in the fragment shader for slices are moved to the vertex shader (Algorithm 5) and interpolated per fragment (Algorithm 6) to reduce the overall computation time.

---

**Algorithm 5:** Vertex shader structure for the rendering of grass blades defined by geometry

---

```

function geometryVertexShader()
begin
  Compute the vertex position in world and camera space (with influence of wind);
  Compute the texture coordinates for the grass blade texture, the density map and
  the color map;
  vertexColor = modulation color per blade; // to make each blade look a bit
  different
  Get the light direction from the current vertex in patch space;
  Compute the vertex normal in patch space, accounting for two-sided lighting;
  luminance = (max(dot(vertexNormal, lightDirection), 0)
    +  $\gamma \cdot \max(\text{dot}(-\text{vertexNormal}, \text{lightDirection}), 0)$ ) . distanceAttenuation;
    // Equation 4.1

  Determine the coordinate frame of the shadow mask space (for the current grass
  blade); // Figure 4.16
  Compute the coordinates of the current vertex in shadow mask space;
  Compute the light direction from the current vertex in shadow mask space;
  Compute the intersection of the light direction vector with the shadow mask
  cylinder;
  Get the shadow mask texture coordinates from this intersection point;
end

```

---

---

**Algorithm 6:** Fragment shader structure for the rendering of grass blades defined by geometry

---

```
function geometryFragmentShader()  
begin  
    Get the density threshold dth of the current grass blade and the local density ld  
    on the terrain;  
    if ld < 0.01 then discard fragment;  
    if dth > ld then discard fragment;  
    Compute  $w_{vs}(d)$  as in Equation 4.14;  
    Compute opacity as in Equation 4.6;  
    if opacity < 0.01 then discard fragment;  
  
    Read values from the blade texture, the terrain color map and the terrain ambient  
    occlusion map;  
    Read the value of the shadow mask and modulate it by the local density;  
    // Equation 4.7  
    fragmentColor =  
    (sceneAmbient . bladeAmbientOcclusion . terrainAmbientOcclusion  
     + lightIntensity . luminance . shadowMaskValue)  
     . bladeTexture . localColor . vertexColor;  
    fragmentAlpha = bladeTextureAlpha . opacity;  
end
```

---

### 4.3.2 Order-independent rendering of semi-transparent quadrilaterals

The rendering order of several semi-transparent quadrilaterals is crucial to avoid visual artifacts. Due to the high number of primitives (geometric grass blades, slices), sorting is impractical for real-time rendering. 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 incorrect pixels are rendered, as in Figure 4.24(a). Sorting of grass blades has to be done by the CPU whenever the camera is moving, requiring excessive processing time. Conversely, alpha testing requires no sorting because the process is binary: the fragment is rendered only if its alpha value satisfies a condition. The most important problem of alpha testing is aliasing, as shown in Figure 4.24(b). Our approach uses blending and alpha testing simultaneously (Figure 4.24(c)). To eliminate the fragments outside a blade, we use alpha testing with a low threshold. This results in a coarse version of the blade shape. Blending refines the transparency process by mixing the borders with the background pixels, thus creating an anti-aliasing effect. Blending artifacts still occur, however, only on the one pixel wide borders, hardly visible in most cases.

Aliasing can be further decreased with the use of *multisampling* but it requires a high fillrate. It is a tradeoff between quality and speed. We enable the mapping of fragment alpha values to coverage values. No sorting of the grass blades is needed, wrong borders totally disappear and grass blades do not present aliasing. Nevertheless, a high precision

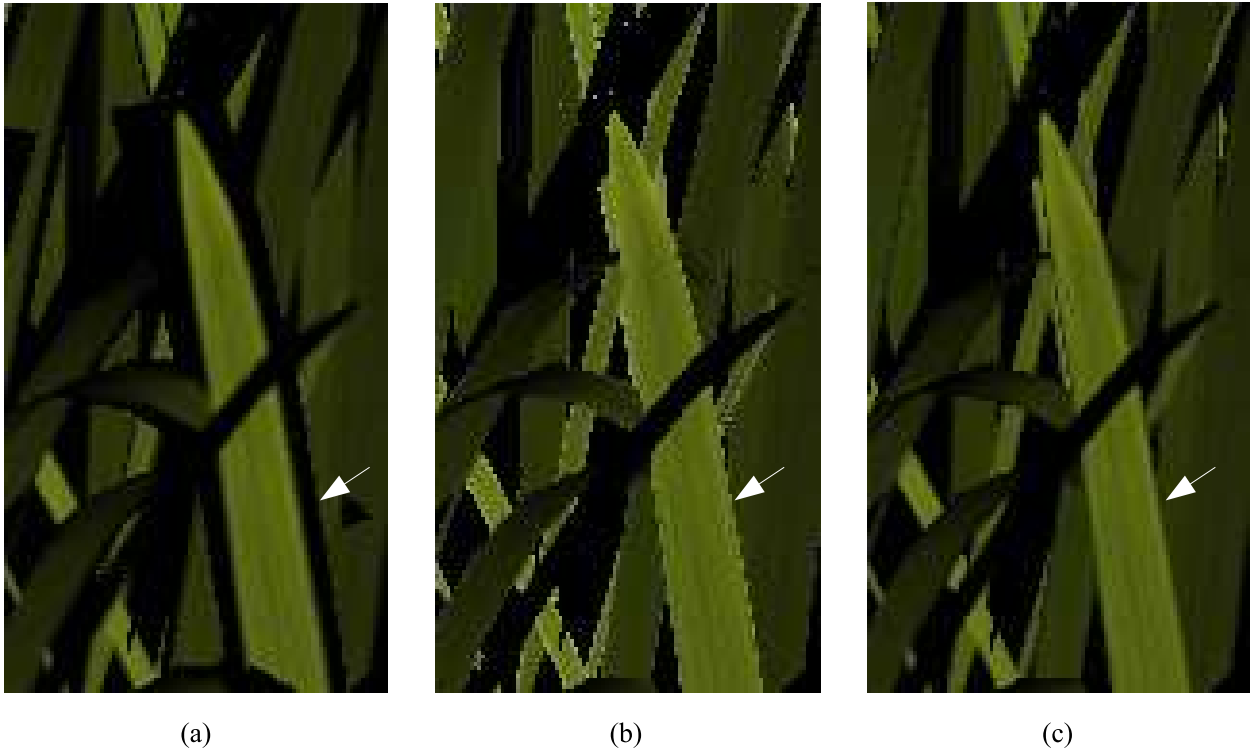


Figure 4.24: Comparison between three methods to process semi-transparency. The white arrows point to a zone where the advantages and drawbacks particularly appear. (a) Simple alpha blending. Sorting for rendering has to be done. (b) Alpha testing, with a high aliasing effect. (c) Both methods, reducing the aliasing and the need for sorting.

of the multisample buffer is needed to make the dithering effect unnoticeable (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 paper make use of multisampling.

### 4.3.3 A custom filter to create the mipmaps pyramid of a semi-transparent texture

Rendering of polygons using a semi-transparent texture is often dedicated to the rendering of faraway objects (billboards), for optimization purpose. However, aliasing artifacts occur when applying simple bilinear filtering. Mipmapping improves the result but introduces a new issue: when creating the alpha channel data for each image of the mipmap pyramid using the regular box filter, the alpha images tend to turn black. Then, in the case of horizontal slices, macro-cells tend to totally disappear when they are faraway from the viewer. Usually, a texel at a given mipmap level is created by taking the average of the four corresponding texels of the previous level. Consequently the alpha channel includes too many black texels. The color channel also gets darker if the objects of the semi-transparent image are in front of a black background. That is why we propose a custom filter. This filter can be applied to many image based rendering methods that make use of semi-transparent textures.

Let  $(R_l[j][i], G_l[j][i], B_l[j][i], A_l[j][i])$  be the *RGB* color and alpha value of the texel  $(i, j)$  of the image ( $j^{th}$  row,  $i^{th}$  column), at level  $l$  of the mipmap pyramid. Let  $C_l[j][i]$  be one of the color component. The regular box filter gives the color of a texel at level  $l + 1$  as follows:

$$\begin{cases} C_{l+1}[j][i] = ( C_l[2j][2i] + C_l[2j][2i+1] + C_l[2j+1][2i] + C_l[2j+1][2i+1] ) / 4 \\ A_{l+1}[j][i] = ( A_l[2j][2i] + A_l[2j][2i+1] + A_l[2j+1][2i] + A_l[2j+1][2i+1] ) / 4 \end{cases} \quad (4.15)$$

Our filter takes the alpha channel into account to generate the color component of the texel. We generate the alpha component of the texel using the maximum alpha value of the four previous level samples.

$$\left\{ \begin{array}{l} a_1 = A_l [2j][2i] \\ a_2 = A_l [2j][2i + 1] \\ a_3 = A_l [2j + 1][2i] \\ a_4 = A_l [2j + 1][2i + 1] \end{array} \right. \quad (4.16)$$

$$sum = \left\{ \begin{array}{ll} 1 & \text{if } a_1 + a_2 + a_3 + a_4 = 0, \\ a_1 + a_2 + a_3 + a_4 & \text{otherwise} \end{array} \right. \quad (4.17)$$

$$\left\{ \begin{array}{l} C_{l+1}[j][i] = (a_1.C_l[2j][2i] + a_2.C_l[2j][2i + 1] + a_3.C_l[2j + 1][2i] + a_4.C_l[2j + 1][2i + 1]) / sum \\ A_{l+1}[j][i] = max(A_l[2j][2i], A_l[2j][2i + 1], A_l[2j + 1][2i], A_l[2j + 1][2i + 1]) \end{array} \right. \quad (4.18)$$

## 4.4 Results

Our level of detail approach allows the rendering of large grass terrains in real-time. Our implementation using the *OpenGL Shading Language* for the shaders, on a 3 GHz *Pentium D* computer with a *nVidia GeForce 7800 GTX* graphics card, renders a football field with



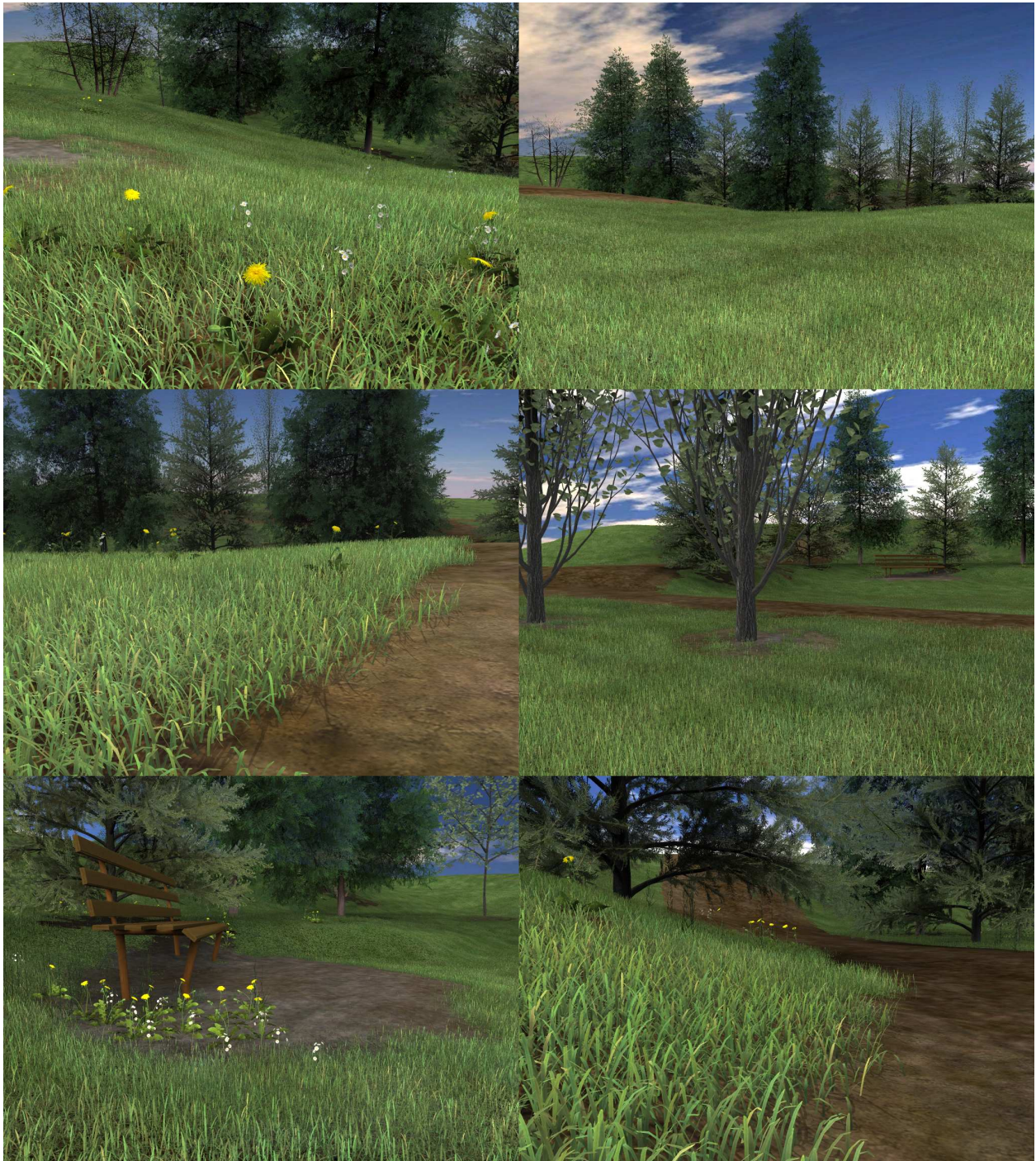


Figure 4.25: Screenshots of the park demo. The grass terrain is modeled using user-defined height and grass density maps. These images are rendered at 10 to 16 frames per second using a *nVidia GeForce 7800 GTX* graphics card.

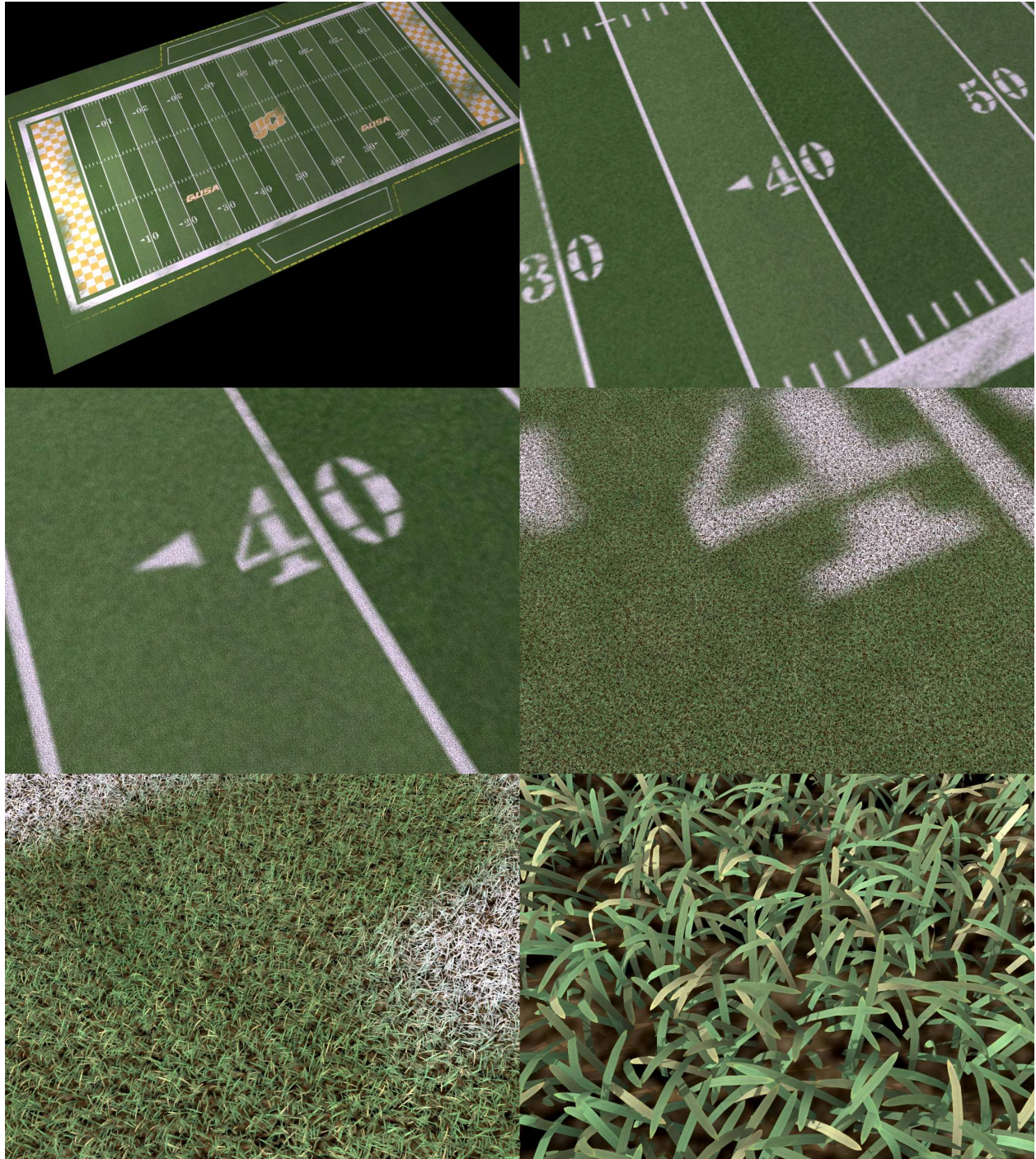


Figure 4.26: Football field of 627 million virtual grass blades seen at different distances.

about 627 million virtual grass blades (Figure 4.26) at a rate of 20 to 250 frames per second (fps) at a resolution of  $1024 \times 768$  with 4X anti-aliasing. The rendering speed varies in this range as a function of the camera position and orientation. When the terrain is seen entirely, the frame rate is maximal. When the scene is seen from the football player's height, the frame rate is about 80 fps. Frame rates of about 20 fps are obtained when grass is seen at 5 centimeters from the ground and horizontally, which is an unusual view angle.

The park demo (Figures 4.1 and 4.25) shows the integration of natural elements like trees on a non flat terrain with user-defined grass density. Most of the images of Figure 4.25 are rendered at 10 to 16 fps (14 to 22 fps if trees are not displayed, as they represent more than 600,000 alpha blended polygons). When the scene is seen from the eyes of a viewer walking in the park, the rendering speed is from 15 to 55 fps. When the park is seen as a whole, the frame rate is about 120 fps. The speed is relatively slower compared to the rendering of flat terrains, mostly because curved terrains significantly reduce the size of the macro-cells and hence increase the amount of computation. Adding a second graphics card (as in *nVidia SLI*), an increase of speed of 10% to 45% was achieved.

Our level of detail management allows high frame rates for faraway views and for views at human height. The rendering is slowest when the vertical slices cover a large surface of the rendered window. This case happens when grass is observed horizontally from low altitude. The BTF slices rendering speed is dependent on the number of rasterized fragments due to the high depth complexity (several fragments are processed per pixel).



Figure 4.27: 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.

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  $512 \times 64$  and  $512 \times 512$  for the horizontal slice. The total amount of data is 45 megabytes without compression, but we use texture compression at the GPU level to reduce memory usage and increase the texture fetch speed. Only 2 seconds are needed at run-time to load the BTF data from disk.

An interesting advantage of our level 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 obtain in 3D applications due to the expensive computations. Our approach allows dynamic lighting and shadowing. The light position can be changed as desired, as shown in Figure 4.27.

# CHAPTER 5: RENDERING TREES IN REAL-TIME WITH INDIRECT LIGHTING

## 5.1 Introduction

Real-time rendering of trees has attracted attention due to its frequent use in video games, flight simulators, ecosystem simulations, etc. However, the computational complexity of tree rendering has always been an obstacle. Geometric complexity is due to the large number of leaves in a tree. Lighting complexity is due to the numerous interactions between leaves and branches, and to the simulation of reflectance properties of leaves that are constituted of multiple layers. Multiple approaches have been used to increase the rendering speed, thereby reducing the overall quality of the rendered images.

Our goal is to design a lighting model that allows real-time rendering of trees with convincing indirect lighting (Figure 5.1). Rendering trees with direct lighting only is subject to undesirable noise, which can be drastically reduced by adding indirect lighting, whose computation is an expensive process. We propose approximations that minimally affect the overall quality of low frequency lighting but increase the rendering speed. High frequency details are added afterwards to enhance realism.



Figure 5.1: (a) Photograph of a tree. (b) Rendering with our method.

Rather than defining a model to mimic lighting of real trees, we work at a lower level by modeling the spatial distribution of leaves and by assigning them probabilistic properties. Using this model, we derive simple equations that are efficient to be processed on graphics hardware and that allow real-time rendering.

We start by giving an overview of our rendering method. We then detail our model for direct lighting, followed by indirect lighting and shadows. Next we give some implementation hints for fast rendering. We finish this chapter by giving some results.

## 5.2 Overview

Our method considers the leaves of a tree as spatially distributed and oriented according to probabilistic laws, these leaves being contained in an envelope defined by a discrete function of direction. It allows the computation of one bounce diffuse inter-reflection within trees, direct sun and sky lighting, while fully exploiting the performances of the GPU to achieve interactivity. In this section, we detail the construction of the envelope and then present the considered lighting environment. Next, we present the materials assigned to the leaves as well as a function giving the attenuation of light through leaf layers.

### 5.2.1 Constructing the tree envelope

We model a tree with an envelope of volume  $V$  that encloses its leaves. This envelope contains a set of uniformly distributed leaves with a density  $\rho$ , a constant area  $A$  and uniformly distributed normals. Rather than defining the envelope shape once for the whole tree, we use a leaf specific approximation of this envelope by assigning distances  $s_{max}(P, \omega_j)$  between each leaf's center point  $P$  and the envelope for  $N_{dir}$  sampling directions  $\omega_j$  (Figure 5.2(a)) uniformly distributed over the unit sphere  $\Omega$ . We detail how we evaluate these distances in Section 5.6. The set of sampling directions  $\omega_j$  is the same for all leaves rather than being relative to the leaf orientations. This allows the evaluation of incoming radiance per tree rather than per leaf, and is more efficient at runtime. With each direction  $\omega_j$  is associated



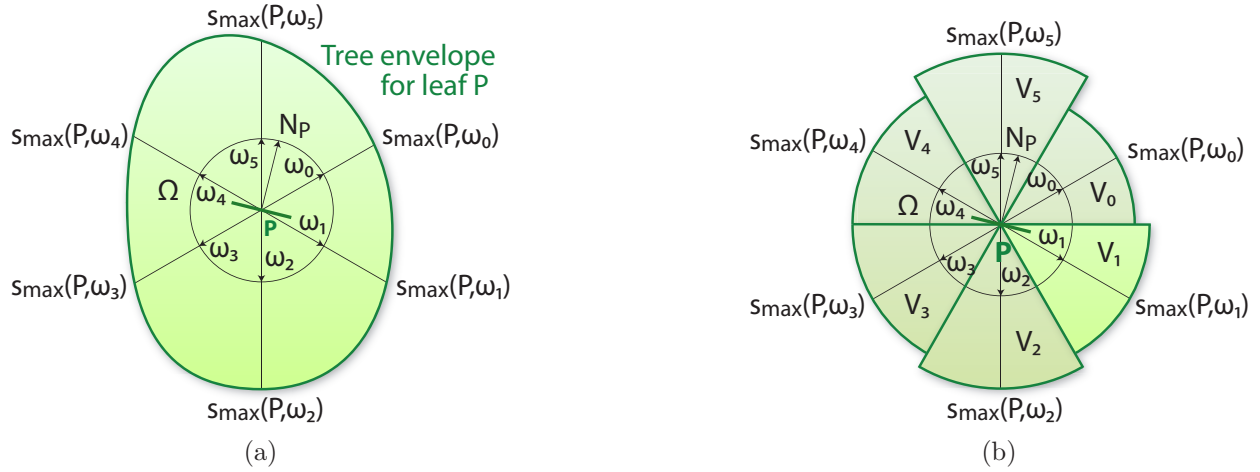


Figure 5.2: (a) Parametrization of the envelope of a tree from a leaf's center point  $P$ . (b) Equivalent set of volumes  $V_j$ .

a sub-volume  $V_j$  of the volume  $V$ , with a solid angle  $\Omega_j$ , having a pyramidal shape starting from  $P$  and directed by  $\omega_j$  (Figure 5.2(b)). We have  $V = \sum_{j=1}^{N_{dir}} V_j$ . The number of leaves present in each volume  $V_j$  is

$$N_{leaves_j} = \rho V_j = \rho \frac{4\pi}{N_{dir}} \frac{(s_{\max}(P, \omega_j))^3}{3} \quad (5.1)$$

## 5.2.2 Lighting environment

Trees are most of the time present outdoors. We then define the lighting environment as follows: an environment light source (sky and reflection of sky light off the ground) and a directional light source (sun). Our approach is totally dynamic: radiance from the sky and the sun, and position of the sun can be modified as desired.

Given a leaf centered at  $P$ , its front side normal is  $N_P$ .  $\omega_o$  is the normalized vector from the leaf to the viewer.  $L_o(P, \omega_o)$  is the radiance reaching the viewer and is equal to the following sum (Figure 5.3):

$$L_o(P, \omega_o) = L_{o_{sky}}(P, \omega_o) + L_{o_{sun}}(P, \omega_o) + L_{o_{ind}}(P, \omega_o) \quad (5.2)$$

$L_{o_{sky}}$  is the outgoing radiance due to direct lighting from the sky and the reflection of skylight off the ground (Section 5.3.1).  $L_{o_{sun}}$  is the outgoing radiance due to direct sun lighting (Section 5.3.2). Finally,  $L_{o_{ind}}$  is the outgoing radiance due to indirect lighting from the sun, after one bounce off neighbor leaves (Section 5.4).

### 5.2.3 Leaf materials

When computing low-frequency lighting, we assume the leaf reflectance properties to be Lambertian for both sides. Therefore, we perform lighting computations for each leaf rather than each pixel as we consider the variation of irradiance along a leaf as negligible. We then introduce variations along the leaf using a texture that modulates the reflectance. We make use of a normal map to model a per-pixel specular component and a thickness map to modulate the transmitted light. We consider four parameters for the leaf material model: the reflectance  $\rho_{r_f}$  of the front face (the same side as  $N_P$ ), the reflectance  $\rho_{r_b}$  of the back face, the transmittance  $\rho_{t_f}$  for light entering the front face, and the transmittance  $\rho_{t_b}$  for



Figure 5.3: The different lighting components. From left to right: direct lighting due to the sky and the light reflected off the ground ( $L_{o_{sky}}$ ), direct lighting due to the sun ( $L_{o_{sun}}$ ), indirect lighting due to the sun ( $L_{o_{ind}}$ ) and sum of all components ( $L_o$ ).

light entering the back face. When observing real trees, their front face is often reflective with a darker color, their back face being matte with a lighter color.

The outgoing radiance  $L_o$  in direction  $\omega_o$  from the current leaf centered at  $P$ , due to the incident radiance  $L$ , is given by

$$L_o(P, \omega_o) = \sum_{j=1}^{N_{dir}} \int_{\Omega_j} F(P, \omega_o, \omega) L(P, \omega) d\omega \quad (5.3)$$

$$F(P, \omega_o, \omega) = \begin{cases} \frac{\rho_{rf}}{\pi} M(N_P \cdot \omega) + \frac{\rho_{tb}}{\pi} M(-N_P \cdot \omega) & \text{if } N_P \cdot \omega_o \geq 0 \\ \frac{\rho_{tf}}{\pi} M(N_P \cdot \omega) + \frac{\rho_{rb}}{\pi} M(-N_P \cdot \omega) & \text{if } N_P \cdot \omega_o < 0 \end{cases} \quad (5.4)$$

$F$  being the cosine-weighted BSDF and  $M(x) = \max(x, 0)$ .

## 5.2.4 Attenuation function

Leaves of a tree are occluded by other leaves. This results in an attenuated outgoing radiance from each leaf. These shadows can be characterized as high frequency since they are sharp and numerous inside a tree. We use a function that gives the average attenuation of light to smooth the high frequency shadows, which reduces the aliasing effects for distant trees. Our hypothesis of leaf uniform spatial distribution results in an attenuation function that only depends on the thickness  $s_{max}(P, \omega_j)$  of the leaf layer between a lit point  $P$  and the light source of radiance  $L(\omega_j)$  in direction  $\omega_j$ . Based on the density of leaves  $\rho$  and the area of each leaf  $A$ , we can find the optical thickness  $\tau$  and define the attenuated radiance  $L(P, \omega_j)$

at point  $P$  as follows:

$$L(P, \omega_j) = e^{-\tau \cdot s_{max}(P, \omega_j)} L(\omega_j) \quad \text{with } \tau = \frac{A}{2} \rho \quad (5.5)$$

This function is similar to the one derived by Max [MO95] for a beam of light traversing a medium containing a high number of spherical particles. However, we replace the spherical particles with randomly oriented leaves of area  $A$  with uniform distribution of orientation. Therefore our optical thickness  $\tau$  uses the average projected area  $A/2$  rather than  $A$ .

## 5.3 Direct Lighting

In this section, we present how to adapt the traditional rendering equations expressing direct lighting to our probabilistic representation of trees.

### 5.3.1 Light from the sky and the ground

We propose a simple method to compute direct lighting of leaves due to the sky light and its reflection off the ground. Rather than using an ambient occlusion approach, which considers the radiance from the environment constant for every direction, we handle variable radiance

from a low-frequency environment. We make use of numerical integration as follows:

$$L_{o_{sky}}(P, \omega_o) \approx \frac{4\pi}{N_{dir}} \sum_{j=1}^{N_{dir}} F(P, \omega_o, \omega_j) e^{-\tau \cdot s_{max}(P, \omega_j)} L_{sky}(\omega_j) \quad (5.6)$$

$L_{sky}(\omega_j)$  is the radiance from the sky in direction  $\omega_j$ . This radiance is attenuated by the function of Equation 5.5.  $L_{sky}(\omega_j)$  has to be determined for each direction  $\omega_j$  of each tree. To this end, we average the pixels belonging to the solid angle  $\Omega_j$  from a low resolution sky environment map. This process is fast and has to be performed only when lighting conditions change. We compute the radiance for the lower hemisphere (reflection of the sky light off the ground) by modulating the radiance of the higher hemisphere with the average reflectance of the ground. By modulating the incoming radiance, it is possible to simulate the occlusions by other trees, by buildings or any other large objects. Animating the leaves by rotating their normal is made possible by the cosine terms present in the  $F$  function that smooth the variations of outgoing radiance.

### 5.3.2 Directional Light Source

We define the sun as a directional light source of direction  $\omega_l$  and irradiance  $E_{sun}$ . In most real-time rendering methods, the outgoing radiance relies only on the angle between the light incidence direction and the normal to the lit surface. The visibility term can be handled using different shadowing techniques. However, we are interested in the overall

shadowing of the tree that can be seen at any distance without aliasing. We then use the attenuation function of Section 5.2.4 to estimate the overall behavior of shadows and we add high frequency shadows afterwards. The distance  $s_{max}(P, \omega_l)$  for an arbitrary light direction  $\omega_l$  is unknown as Equation 5.5 can be evaluated only for the sample directions  $\omega_j$ . We can get an estimate of the shape of the tree for any direction  $\omega_l$  by interpolating the  $s_{max}(P, \omega_j)$  values. Alternatively, any direction  $\omega_l$  can be assigned the distance  $s_{max}(P, \omega_j)$  where  $\omega_j$  is the closest sample direction to  $\omega_l$ . This results in a very coarse approximation (Figure 5.4(a)), even when modulating the  $s_{max}$  distance by  $(\omega_j \cdot \omega_l)$  (Figure 5.4(b)). These two functions are subject to discontinuities that result in sudden changes of lighting when the sun moves. Choosing the maximum of  $(\omega_j \cdot \omega_l) \cdot s_{max}(P, \omega_j)$  among all  $j$  removes the discontinuities (Figure 5.4(c)). However, this function presents important variations of lighting when the sun moves. We finally retain the function of Figure 5.4(d), defined as follows:

$$\widetilde{s_{max}}(P, \omega_l) = \frac{\sum_{j=1}^{N_{dir}} M(\omega_l \cdot \omega_j) \cdot s_{max}(P, \omega_j)}{\sum_{j=1}^{N_{dir}} M(\omega_l \cdot \omega_j)} \quad (5.7)$$

This function approximates the envelope shape using a set of cosine lobes contained in the hemisphere directed by  $\omega_l$  (non linear due to the  $M$  function). This approximation is robust even for leaves close to the tree envelope. Splines could be used as well, however they would increase drastically the computation time. Spherical harmonics would provide a smooth function for  $\widetilde{s_{max}}$ , but would require the evaluation of polynomials and cosine functions, which is time consuming for real-time applications.

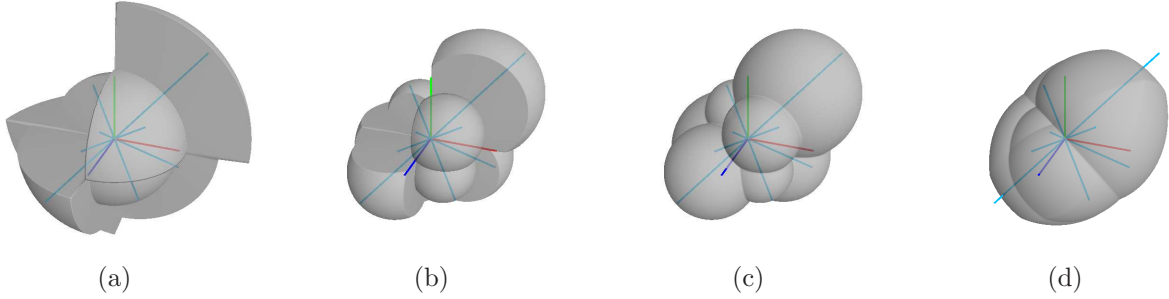


Figure 5.4: Four possible functions to estimate the shape of the tree envelope for a leaf centered at P. The (red,green,blue) lines represent the coordinate frame of the leaf. The cyan lines are the vectors  $\omega_j$  scaled by  $s_{max}(P, \omega_j)$  with  $N_{dir} = 8$ . The distance of the gray surface from P represents the  $\widetilde{s}_{max}(P, \omega_l)$  distances for all possible  $\omega_l$ .

The final equation of outgoing radiance due to the sun light becomes

$$L_{o_{sun}}(P, \omega_o) = F(P, \omega_o, \omega_l) e^{-\tau \cdot \widetilde{s}_{max}(P, \omega_l)} E_{sun} \quad (5.8)$$

## 5.4 Indirect Lighting

The contribution of indirect lighting is of high importance within a set of leaves due to the large number of reflections and transmissions of light that take place in this set (Figure 5.5). Our approach is based on probabilistic models to get an estimate of indirect lighting due to one bounce of sun light that occurs in the neighborhood of a leaf. We do not manage more bounces due to the high extra processing time and the low difference in result. We also ignore indirect lighting due to sky light. Its contribution is low (6% in the case of Figure



5.16(c)), is almost uniform, and would require expensive computation. We can add an offset to the resulting outgoing radiance to approximate this component but it is not necessary. When there is no sunlight during overcast days, light is still reaching the bottom of the set of leaves since the ground reflects light from the sky.

For each leaf, we want to compute the lighting contribution of all neighbor leaves. The naive approach would be a simple sum of the outgoing radiance of each neighbor leaf. Unfortunately, it requires a large amount of computation, particularly due to the occlusions between the currently lit leaf and the neighbor leaves, and the occlusions between the neighbor leaves and the environment. To overcome this problem, we look for some invariants that make our equations simpler and thereby faster to compute.

We first present our integration scheme that isolates some invariants. We then show how some parts of the equations can have analytical solutions or estimations. We finally express the estimation of incoming radiance from the sun onto neighbor leaves.

### 5.4.1 Integration Scheme

For lighting purpose, we consider the solid angles  $\Omega_j$  of the volumes  $V_j$  (Figure 5.2(b)) small enough to place all neighbor leaf centers  $P'$  contained in  $V_j$  on the rays originating at  $P$  and of direction  $\omega_j$ . The attenuation term between the current leaf and a neighbor leaf depends only on the distance between the leaves (Section 5.2.4). Therefore we subdivide each volume  $V_j$  into a large number  $N_{s_j}$  of thin concentric slices  $V_{j,i}$  of radius  $s_i$  and of



Figure 5.5: Tree rendering (a) without indirect lighting, (b) with indirect lighting.

thickness  $\Delta s_j = s_{max}(P, \omega_j)/N_{s_j}$  with  $s_i = i \cdot \Delta s_j$  (Figure 5.6). The attenuation term is supposed to be constant inside these slices.

Equation 5.3 uses an integral over the incidence directions  $\omega$ . We start by transforming it to a sum of integrals over the leaf surfaces, each leaf having the same area  $A$ . We consider the leaves small compared to the distance from  $P$  to  $P'$ . The integrand is then considered to be constant for each point of a given neighbor leaf centered at  $P'$ . The outgoing radiance due to reflection and transmission of sun light by neighbor leaves becomes:

$$L_{o_{ind}}(P, \omega_o) = A \sum_{j=1}^{N_{dir}} \sum_{i=1}^{N_{s_j}} \sum_{P' \in V_{j,i}} F(P, \omega_o, \widehat{PP'}) L_{o_{sun}}(P', -\widehat{PP'}) V(P, P') \frac{|N_{P'} \cdot (-\widehat{PP'})|}{\|PP'\|^2} \quad (5.9)$$

$\widehat{PP'}$  is the normalized  $PP'$  vector.  $N_{P'}$  is the normal to the leaf centered at  $P'$ . The number of leaves contained in the slice  $V_{j,i}$  (such that  $P' \in V_{j,i}$ ) is

$$N_{leaves_{j,i}} = \rho \frac{4\pi}{N_{dir}} s_i^2 \Delta s_j \quad (5.10)$$

where  $\Delta s_j$  is small.

As presented earlier, the solid angles  $\Omega_j$  are small enough to consider  $PP'$  as similar to  $\omega_j$ . We also consider the slices  $V_{j,i}$  as very thin, hence  $\|PP'\| = s_i$ . We can then approximate the visibility function  $V(P, P')$  by the attenuation term  $e^{-\tau \cdot s_i}$ . By applying the last three properties, we can rewrite Equation 5.9 as follows:

$$L_{o_{ind}}(P, \omega_o) = A \sum_{j=1}^{N_{dir}} F(P, \omega_o, \omega_j) L_{o_{sample}}(P, j) \quad (5.11)$$

$$L_{o_{sample}}(P, j) = \sum_{i=1}^{N_{s_j}} \frac{e^{-\tau \cdot s_i}}{s_i^2} L_{o_{slice}}(P, j, i) \quad (5.12)$$

$$L_{o_{slice}}(P, j, i) = \sum_{P' \in V_{j,i}} L_{o_{sun}}(P', -\omega_j) |N_{P'} \cdot (-\omega_j)| \quad (5.13)$$

$L_{o_{sample}}(P, j)$  corresponds to the outgoing radiance of all leaves inside  $V_j$ .  $L_{o_{slice}}(P, j, i)$  represents the outgoing radiance of all leaves inside the slice  $V_{j,i}$ .  $L_{o_{sun}}$  is the total outgoing radiance from neighbor leaves when illuminated by the attenuated irradiance of the sun  $E_{sun}(P')$ :

$$L_{o_{sun}}(P', -\omega_j) = F(P', -\omega_j, \omega_l) E_{sun}(P') \quad (5.14)$$

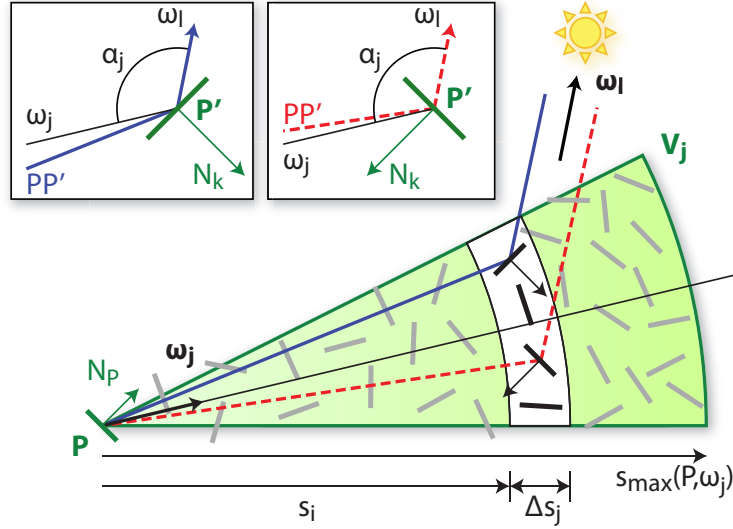


Figure 5.6: Subdivision of a volume  $V_j$  into slices of radius  $s_i$  and of thickness  $\Delta s_j$ . The blue line is an example of reflected ray on a leaf of the slice (left inset), the red dashed line is an example of transmitted ray through a leaf of the slice (right inset). For all leaves inside  $V_j$ ,  $\alpha_j$  is the constant angle between  $\omega_l$  and  $\omega_j$ .

### 5.4.2 Evaluation of the sums

The previous equations cannot be efficiently computed at runtime. We want to remove as many sums as possible. Given a leaf centered at  $P' \in V_{j,i}$ , we define its relative position as  $PP' = s_i \cdot \omega_j$ . Rewriting Equation 5.13 using the latter expression, Equation 5.4, and

developing Equation 5.14 gives

$$\begin{aligned}
L_{O_{slice}}(P, j, i) = E_{sun}(P + s_i \cdot \omega_j) \times \\
\left[ \frac{\rho_{r_f}}{\pi} \sum_{P' \in V_{j,i}} M(N_{P'} \cdot \omega_l) \cdot M(N_{P'} \cdot (-\omega_j)) \right. \\
+ \frac{\rho_{t_f}}{\pi} \sum_{P' \in V_{j,i}} M(N_{P'} \cdot \omega_l) \cdot M(-N_{P'} \cdot (-\omega_j)) \\
+ \frac{\rho_{t_b}}{\pi} \sum_{P' \in V_{j,i}} M(-N_{P'} \cdot \omega_l) \cdot M(N_{P'} \cdot (-\omega_j)) \\
\left. + \frac{\rho_{r_b}}{\pi} \sum_{P' \in V_{j,i}} M(-N_{P'} \cdot \omega_l) \cdot M(-N_{P'} \cdot (-\omega_j)) \right] \quad (5.15)
\end{aligned}$$

Only the normal  $N_{P'}$  is variable inside each of the sum. Appendix B gives details on the estimation of these sums using the hypothesis of uniform distribution of leaf orientation. We can use this estimation to rewrite Equation 5.15 as follows:

$$\begin{aligned}
L_{O_{slice}}(P, j, i) = E_{sun}(P + s_i \cdot \omega_j) \rho \frac{4\pi}{N_{dir}} s_i^2 \Delta s_j \times \\
\left[ \frac{\rho_{r_f} + \rho_{r_b}}{\pi} \left( \frac{1}{6\pi} (-\alpha_j \cos \alpha_j + \sin \alpha_j) \right) \right. \\
\left. + \frac{\rho_{t_f} + \rho_{t_b}}{\pi} \left( \frac{1}{6\pi} ((\pi - \alpha_j) \cos \alpha_j + \sin \alpha_j) \right) \right] \quad (5.16)
\end{aligned}$$

with  $\alpha_j \in [0, \pi]$  as the angle between  $\omega_j$  and  $\omega_l$ . Substituting Equation 5.16 into Equation 5.12 yields

$$L_{O_{sample}}(P, j) = G_1(j, \omega_l) \sum_{i=1}^{N_{s_j}} e^{-\tau \cdot s_i} E_{sun}(P + s_i \cdot \omega_j) \Delta s_j \quad (5.17)$$

with

$$G_1(j, \omega_l) = \frac{2\rho [\bar{\rho}(\sin \alpha_j - \alpha_j \cos \alpha_j) + \pi(\rho_{t_f} + \rho_{t_b}) \cos \alpha_j]}{3\pi N_{dir}} \quad (5.18)$$

$$\bar{\rho} = \rho_{r_f} + \rho_{r_b} + \rho_{t_f} + \rho_{t_b} \quad (5.19)$$

$N_{s_j}$  has to be large to guarantee an accurate value of  $L_{o_{sample}}$  (Equation 5.17).  $\Delta s_j$  is inversely proportional to  $N_{s_j}$ . Therefore, by using the following relation,

$$\begin{aligned} I(P, j) &= \lim_{N_{s_j} \rightarrow \infty} \sum_{i=1}^{N_{s_j}} e^{-\tau \cdot s_i} E_{sun}(P + s_i \cdot \omega_j) \Delta s_j \\ &= \int_0^{s_{max}(P, \omega_j)} e^{-\tau \cdot s} E_{sun}(P + s \cdot \omega_j) ds \end{aligned} \quad (5.20)$$

we can get a good approximation for  $L_{o_{sample}}$  by using the integral rather than the discrete sum:

$$L_{o_{sample}}(P, j) \approx G_1(j, \omega_l) \cdot I(P, j) \quad (5.21)$$

### 5.4.3 Evaluation of the irradiance on neighbor leaves

The last step to compute indirect lighting on a leaf  $P$  is to evaluate the irradiance  $E_{sun}(P + s \cdot \omega_j)$  that reaches the neighbor leaves centered at  $P' = P + s \cdot \omega_j$ . This irradiance is attenuated by the layer of leaves separating the neighbor leaf from the direct illumination of the sun.

Using the attenuation function of Section 5.2.4, we can write

$$E_{sun}(P + s.\omega_j) = e^{-\tau.\widetilde{s_{max}}(P+s.\omega_j,\omega_l)} E_{sun} \quad (5.22)$$

However, we cannot afford to evaluate  $\widetilde{s_{max}}(P + s.\omega_j, \omega_l)$  using Equation 5.7 because this evaluation has to be done inside the integral of Equation 5.21. We need an easily integrable function. We cannot use a simple primitive such as a sphere to approximate the tree envelope to easily compute  $\widetilde{s_{max}}$ . The sphere does not fit well an arbitrary tree and the resulting integral is not analytically solvable. We rather propose the following function, more complex in appearance but easily integrable and offering a good fit of the tree envelope:

$$\widetilde{s_{max}}(P + s.\omega_j, \omega_l) = \frac{\sum_{j'=1}^{N_{dir}} [M(\omega_l \cdot \omega_{j'}) \cdot (s_{max}(P, \omega_{j'}) - (\omega_j \cdot \omega_{j'}) \cdot s)]}{\sum_{j'=1}^{N_{dir}} M(\omega_l \cdot \omega_{j'})} \quad (5.23)$$

This function follows the same approach as Equation 5.7 except that it has an additional term to account for the new origin of the distances  $\widetilde{s_{max}}$ . We use this function in Equation 5.22 and solve the integral of Equation 5.21. We finally get the following expression for the contribution of indirect lighting due to the leaves in  $V_j$ :

$$\begin{aligned} L_{o_{sample}}(P, j) &= \frac{G_1(j, \omega_l)}{G_3(j, \omega_l)} E_{sun} \times \\ &\quad \exp\left(-G_2(\omega_l) \cdot \sum_{j'=1}^{N_{dir}} M(\omega_l \cdot \omega_{j'}) \cdot s_{max}(P, \omega_{j'})\right) \times \\ &\quad \left(1 - \exp(-G_3(j, \omega_l) \cdot s_{max}(P, \omega_j))\right) \end{aligned} \quad (5.24)$$

with

$$G_2(\omega_l) = \frac{\tau}{\sum_{j'=1}^{N_{dir}} M(\omega_l \cdot \omega_{j'})} \quad (5.25)$$

$$G_3(j, \omega_l) = \tau \left[ 1 - \frac{\sum_{j'=1}^{N_{dir}} M(\omega_l \cdot \omega_{j'}) \cdot (\omega_j \cdot \omega_{j'})}{\sum_{j'=1}^{N_{dir}} M(\omega_l \cdot \omega_{j'})} \right] \quad (5.26)$$

Equation 5.24 can be finally substituted into Equation 5.11 to evaluate the outgoing radiance at the leaf centered at  $P$  due to indirect lighting. The three above equations represent the main contribution of this paper. The saving that these equations bring in to the whole computation will be shown in the next section.

## 5.5 Shadows

The presence of shadows inside trees and from trees is of high importance for rendered images. Otherwise, no visual clue is available to locate the trees spatially and their illumination looks flat. We can isolate two frequency bands: low and high frequency shadows. Low frequency shadows define the overall shape of the tree, they are very soft and are mainly due to lighting from the sky and to the light interactions between leaves and branches known as indirect lighting. Low frequency shadows have to be pretty accurate since our visual system is sensitive to them. We treated these components in the two previous sections. Our concern in this section is high frequency shadows. They can be more approximate than low frequency shadows since our visual system is less sensitive to them but they need to be present.



No specific work has been done on tree shadows. Traditional methods such as shadow mapping [Wil78] are often used but aliasing due to the presence of many objects in fixed resolution maps is a serious issue. Better shadow mapping methods exist [SD02, AMB07, AMS08] but they are more time expensive and in general harder to implement.

We can isolate 9 types of shadows in scenes with trees made up of branches and leaves and with external objects that are the remaining parts of the scene:

- from leaves to leaves (*a*)
- from leaves to branches (*b*)
- from leaves to external objects (*b*)
- from branches to leaves (*b*)
- from branches to branches (*b*)
- from branches to external objects (*b*)
- from external objects to leaves (*b/c*)
- from external objects to branches (*b/c*)
- from external objects to external objects (*b/c*)

(*a*), (*b*) and (*c*) are three shadowing methods. We are going to present two new methods, for (*a*) and (*b*). The (*c*) method can be traditional shadow mapping or any variant. For (*a*), we present an environment texture based method to project shadows between leaves. For

(b), we present a variation of shadow mapping that handles opacity and approximate soft shadows.

### 5.5.1 Shadows projected onto leaves by other leaves

Accurately projecting the shadow of each leaf onto each other leaf is too expensive for real-time rendering purposes. Since only the presence of these shadows is important and not their accuracy, we reuse the hypotheses of our probabilistic model: uniform density of leaves, uniform distribution of leaf normals and constant area of leaves. We need to find a model that is enough convincing when modifying lighting conditions such as the direction of the sun (Figure 5.22). Additionally, we are trying to handle soft shadows for increased realism.

#### 5.5.1.1 Definition of the shadow mask

We propose an approach based on a texture map surrounding each leaf and representing the set of neighbor leaves occluding the environment. This texture contains the visibility function for each possible incoming light direction: 0 if a neighbor leaf hides the sun and 1 if the sun is visible (Figure 5.7). We call this texture *shadow mask*. However, one shadow mask should be defined for each single leaf, which is too memory intensive. Instead, we use a single generic mask for the whole tree. This mask is generated from the leaf distribution properties

such as the density. The generated shadows will not be exact but will be convincing enough in most cases.

We defined the sun as a directional light source, therefore the direction of the sun is exactly the same for each point of each leaf. If this direction was used to access the shadow mask, the whole tree would be either shadowed or unshadowed. To avoid this issue, we rotate the shadow mask to align it to the coordinate frame of each leaf.

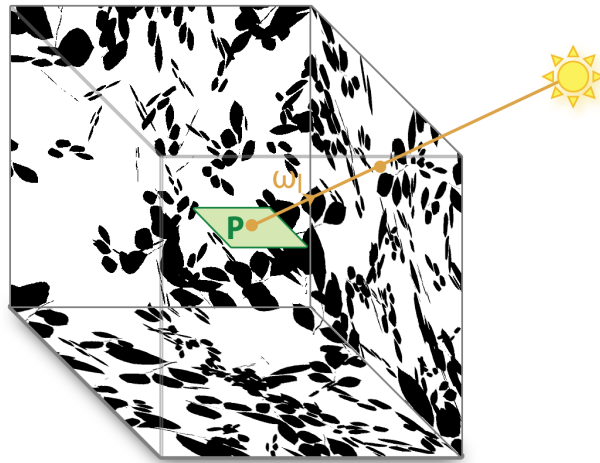


Figure 5.7: Cube shaped shadow mask. The leaf centered at  $P$  is shadowed based on the texture value at the intersection of the ray  $(P, \omega_l)$  and the cube.

Given a leaf of the tree, the number of neighbor leaves casting shadows varies based on the sun direction. This number depends on the location of the current leaf and the thickness of the leaf layer to be traversed. We would need a shadow mask for each possible distance of a leaf to the envelope. Instead, we propose the following method that uses a single mask: the neighbor leaves in the shadow mask are assigned a gray level and a thresholding is performed when rendering. Neighbor leaves close to the current leaf are assigned a dark gray, neighbor

leaves close to the tree envelope are assigned a light gray (Figure 5.8). At rendering time, the distance  $\widetilde{s}_{max}(P, \omega_l)$  of the leaf centered at  $P$  to the envelope in direction  $\omega_l$  is computed as in Equation 5.7 and becomes a threshold: every texel of the shadow mask over this threshold is set to 1, every texel equal to or below the threshold is set to 0, keeping the closest leaves as occluders (Figure 5.9).

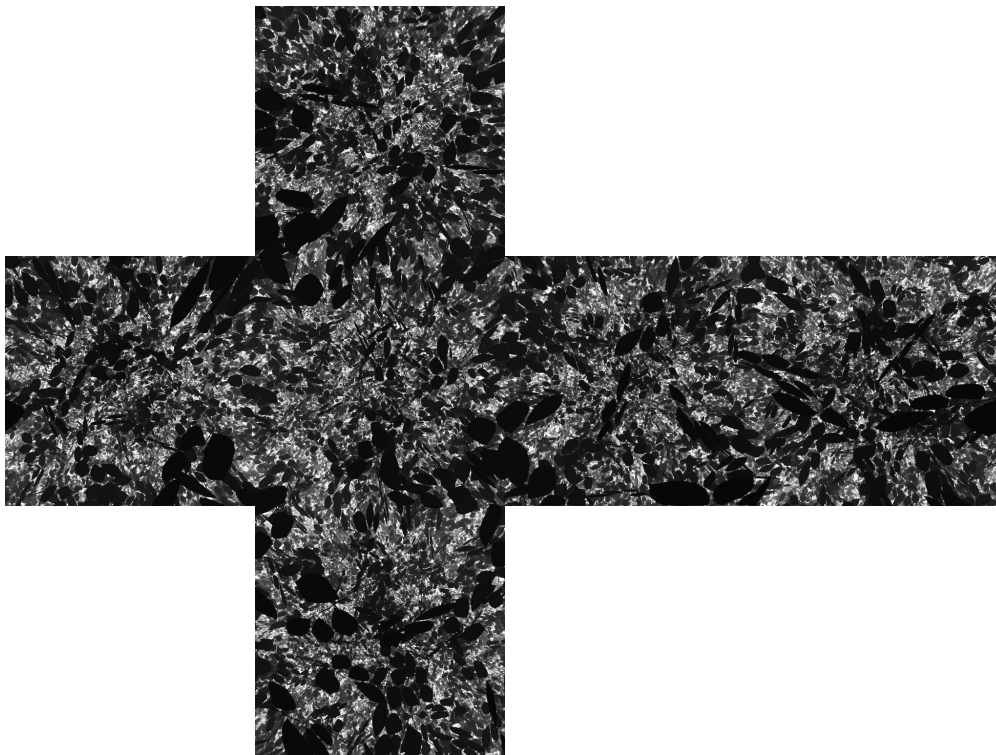


Figure 5.8: Unfolded cube map of the shadow mask.

### 5.5.1.2 Procedural generation of the shadow mask

The content of the shadow mask can be procedurally generated with only the parameters of the leaf distribution and the opacity channel of the leaf texture. We first find the maximum

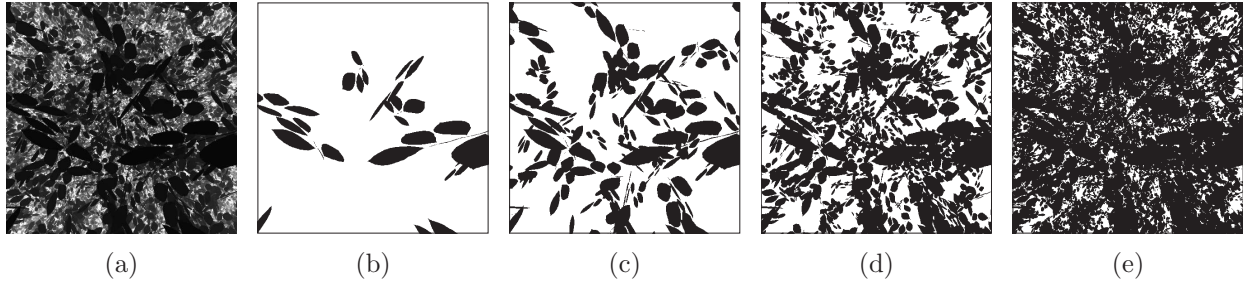


Figure 5.9: Thresholding of the shadow mask. (a) Original shadow mask (one of the faces of the cube). (b)-(e) Processed shadow mask with different increasing threshold values.

possible distance of a leaf to a neighbor by taking the maximum of the  $s_{max}$  values for every leaf and every sample direction. This maximum distance then corresponds to the lightest gray level assigned to leaves in the shadow mask. We then generate a sphere containing uniformly distributed quadrilaterals with uniform distribution of normals (Figure 5.10). The radius of this sphere is the maximum possible distance between two leaves. The quadrilaterals inside the sphere are rendered with a constant gray level proportional to the distance from the sphere center and are semi-transparent, the opacity channel of the leaf texture determines which fragments are rendered. The background of the scene is set to white, corresponding to the absence of occluding leaves. Six cameras are placed at the center of the sphere to render the faces of the shadow mask cube map.

### 5.5.1.3 Multi-layer shadow mask

At rendering time, the sun direction is used to access the shadow mask. However, all points of a leaf are assigned the same value from the shadow mask, resulting in fully occluded or

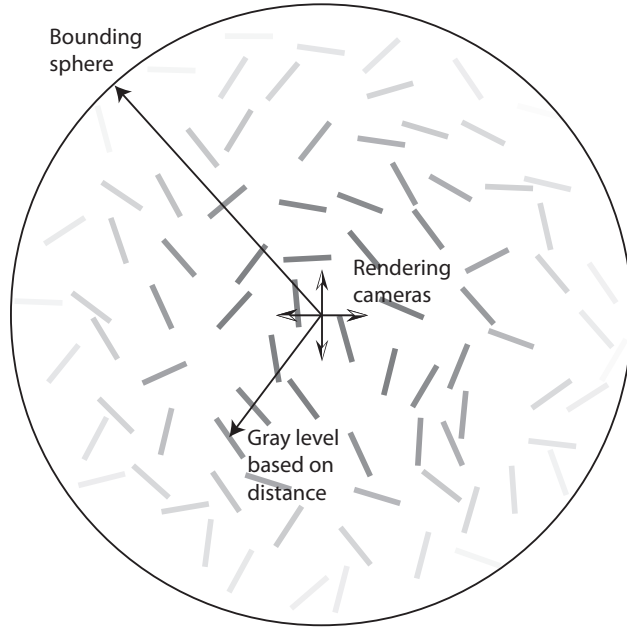


Figure 5.10: Procedurally generated sphere of leaves. Each leaf is assigned a gray level, lighter when farther from the center. The six rendering cameras (for the six faces of the cube) are placed at the center of the sphere.

fully lit leaves. Rays originating from the points of a leaf and in the direction of the sun have to be parallel and intersect different points of the neighbors. We simulate this behavior as shown in Figure 5.11.  $P$  is the current leaf,  $P_1$  and  $P_2$  are two points on the leaf. The rays of direction  $\omega_l$  and origin  $P$ ,  $P_1$  and  $P_2$  should intersect the shadow mask of radius  $t_{23}$  at the points  $B$ ,  $A$  and  $C$  respectively. However, the shadow mask only represents the visibility function for rays starting at  $P$ . For each point of the leaf,  $P_1$  for example, we compute the intersection point of the  $(P_1, \omega_l)$  ray with the sphere of center  $P$  and of radius  $t_{23}$ . This gives the point  $A$  in this case. We then access the shadow mask using the  $PA/||PA|| = PA/t_{23}$  direction. We finally threshold the content of the shadow mask by  $\widetilde{s_{max}}(P, \omega_l)/t_4$  with  $t_4$  the maximum possible distance from a leaf to another one.

Unfortunately, this computation is only valid for neighbor leaves at distance  $t_{23}$  from the current leaf. The resulting effect, when the sun direction  $\omega_l$  changes, is a feeling of flatness, as if the leaves casting shadows were contained in the same plane. We cannot recover the full parallax effect because the shadow mask is only defined for a single point in space. We simulate several layers of leaves casting shadows by partitioning these leaves into several hollow spheres. Since a texture can contain a maximum of four color components (red, green, blue, alpha), we use four spheres. Seen from the current leaf, several layers moving at different speeds give the illusion of parallax when the sun moves. The layers contain leaves from distance 0 to  $t_1$  for the red channel,  $t_1$  to  $t_2$  for the green channel,  $t_2$  to  $t_3$  for the blue channel,  $t_3$  to  $t_4$  for the alpha channel. In our implementation, we use  $t_1 = 0.08 t_4$ ,  $t_2 = 0.15 t_4$  and  $t_3 = 0.3 t_4$ . Figures 5.12 and 5.13 show an example of a multi-layer shadow mask. The intersection points are calculated at the spheres halfway between layer boundaries, at radii  $t_{01}$ ,  $t_{12}$ ,  $t_{23}$  and  $t_{34}$ .

With this method, shadows cast from the farthest layer appear naturally larger than those of the closest layer. However, they should be very soft and lighter due to their distance. We easily simulate this behavior by biasing the mipmap level when accessing the shadow mask. In our implementation, we bias the shadow mask access by two mipmap levels for the green channel, four levels for the blue channel and six levels for the alpha channel. To get the lighter shadows, no additional operation is required: the black and white shadow masks are filtered, creating lighter shadows.

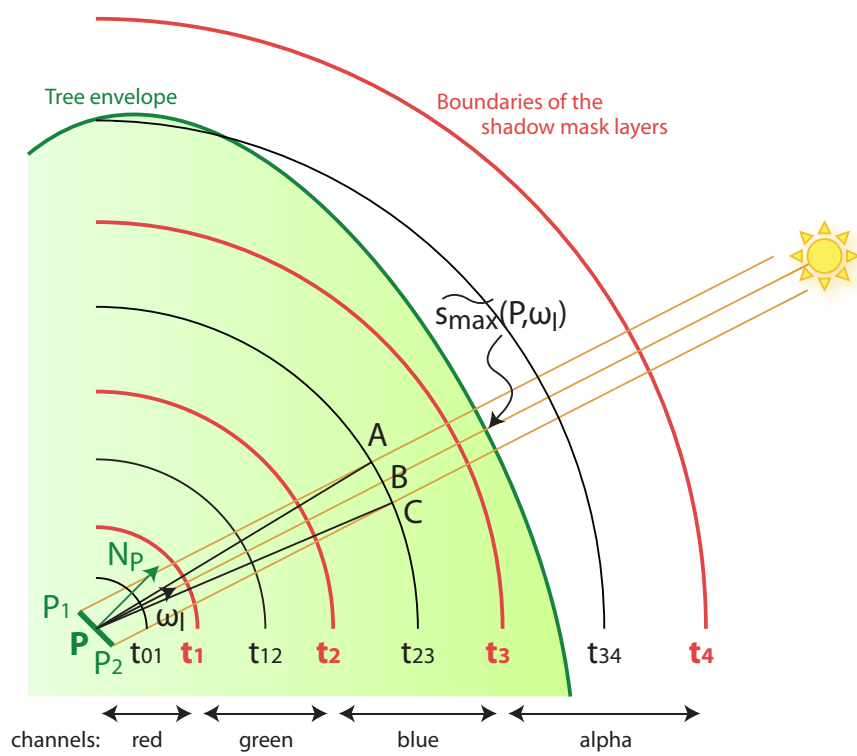


Figure 5.11: Concentric hollow spheres containing subsets of the neighbor leaves.





Figure 5.12: Unfolded cube map of the multi-layer shadow mask (only RGB channels are visible, see Figure 5.13 for the alpha channel).

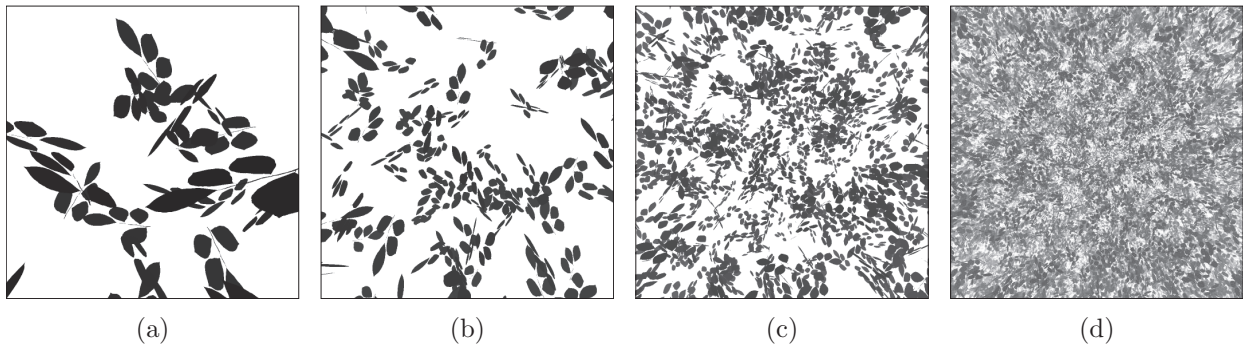


Figure 5.13: The four layers of the shadow mask (one of the faces of the cube). (a) Red channel, with gray levels from 0 to  $t_1/t_4$ . (b) Green channel, with gray levels from  $t_1/t_4$  to  $t_2/t_4$ . (c) Blue channel, with gray levels from  $t_2/t_4$  to  $t_3/t_4$ . (d) Alpha channel, with gray levels from  $t_3/t_4$  to 1.

## 5.5.2 Shadows cast by trees

Shadows cast by trees onto leaves and external objects can be achieved using shadow mapping or any of its variants. However, these methods do not handle semi-opaque objects. Trees projected on ground should not be of uniform darkness, only the part where branches are present should be dark. Shadows are also getting smoother when going farther from the trees. We simulate this behavior with a new method inspired by shadow mapping and projective shadows, which we describe hereafter. This method is made of two passes: shadow map computation and rendering of the final scene.

### 5.5.2.1 Shadow map generation

Rather than generating the shadow map in light space, we generate it in world space. This results in a uniform distribution of shadow map texels along the ground, and makes the opacity accumulation in the shadow map easier to implement. Since trees are placed on a terrain defined most of the time with a height map, we can determine the minimum height of the terrain. We place a horizontal shadow map at this height. The boundaries of this map are determined based on the light direction  $\omega_l$ , the bounding volume of the trees and the minimum height of the terrain (Figure 5.14(a)). A single shadow map is used for all trees if they are close each other. If many scattered trees are present, several shadow maps or an atlas of shadow maps should be used.

The shadow map is recomputed each time the light direction changes. The first step consists in clearing the map with  $(0, 0, 0, 1)$ : the color channels are initialized to black and the alpha channel contains 1, the maximum transparency value (the minimum opacity). When the shadow map is the current render target, the trees casting shadows are rendered but with some modifications. For each vertex  $P$  of the tree, a projection is performed: a point  $P$  is projected to  $P'$  onto the shadow map, the projection direction being  $\omega_l$ . The associated projection matrix is:

$$M_{\omega_l} = \begin{pmatrix} 1 & (-\omega_l.x / \omega_l.y) & 0 & (planeY . \omega_l.x / \omega_l.y) \\ 0 & 0 & 0 & planeY \\ 0 & (-\omega_l.z / \omega_l.y) & 1 & (planeY . \omega_l.z / \omega_l.y) \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.27)$$

$planeY$  is the distance of the shadow map plane from the world origin on the  $Y$  axis. The resulting  $P'P$  vector is stored in the shadow map, only if it is longer than the previous one. All points  $P$  of the tree on the  $(P', \omega_l)$  ray are projected to  $P'$ . Therefore, finding the closest point  $P_{max}$  to the sun on this ray is equivalent to find the  $P'P$  vector of maximum length. Fortunately, current graphics hardware supports blending with the *maximum* operator. There is however a limitation, only positive numbers are supported when rendering. Therefore, we only keep the vertical component  $Y$ , it can be kept strictly positive (Figure 5.15(a)). Comparing the length of  $P'P$  vectors is actually equivalent to compare the  $Y$  component of these vectors.

We want our shadow maps to support variable opacity. To this end, we make use of the alpha channel (Figure 5.15(b)). It contains 1 when starting rendering the shadow map, meaning full transparency. The foliage is then rendered with a constant transparency factor (0.9 in our implementation). Blending is enabled with the *multiply* operator. The successive products of the constant transparency factor become equivalent to an attenuation function expressed as in Equation 5.5. Branches are rendered with a transparency factor of 0. This results in darker shadows where branches are present and lighter shadows in other places. External occluders can also participate to this shadow map with a transparency factor of 0.

The blending operators in OpenGL are configured as follows:

```
glBlendEquationSeparate(GL_MAX, GL_FUNC_ADD);
```

```
glBlendFuncSeparate(GL_ONE, GL_ONE, GL_ZERO, GL_SRC_ALPHA);
```

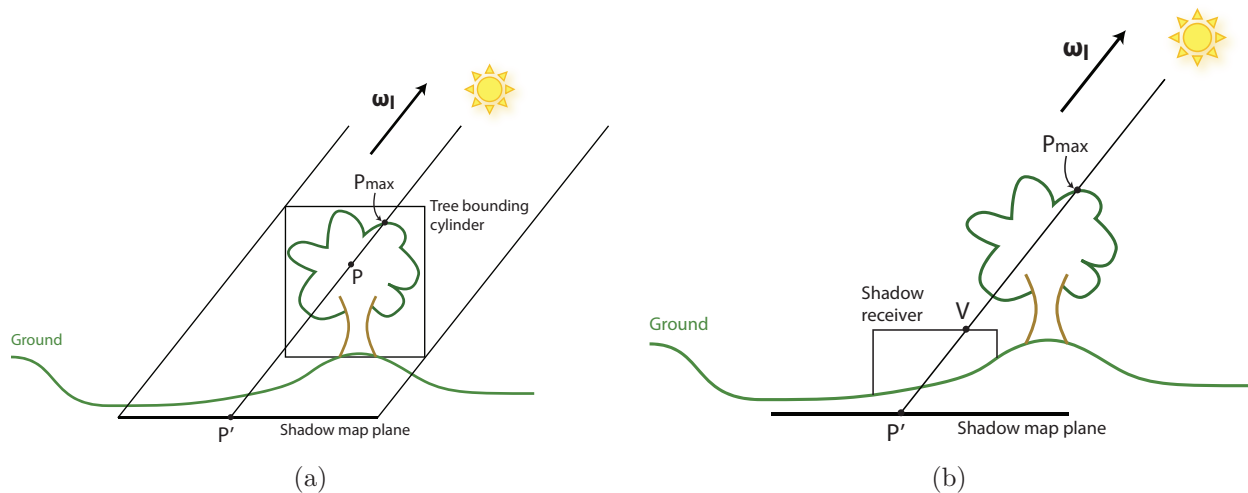


Figure 5.14: (a) Generation of the shadow map. (b) Projection of shadows on a rendered object.

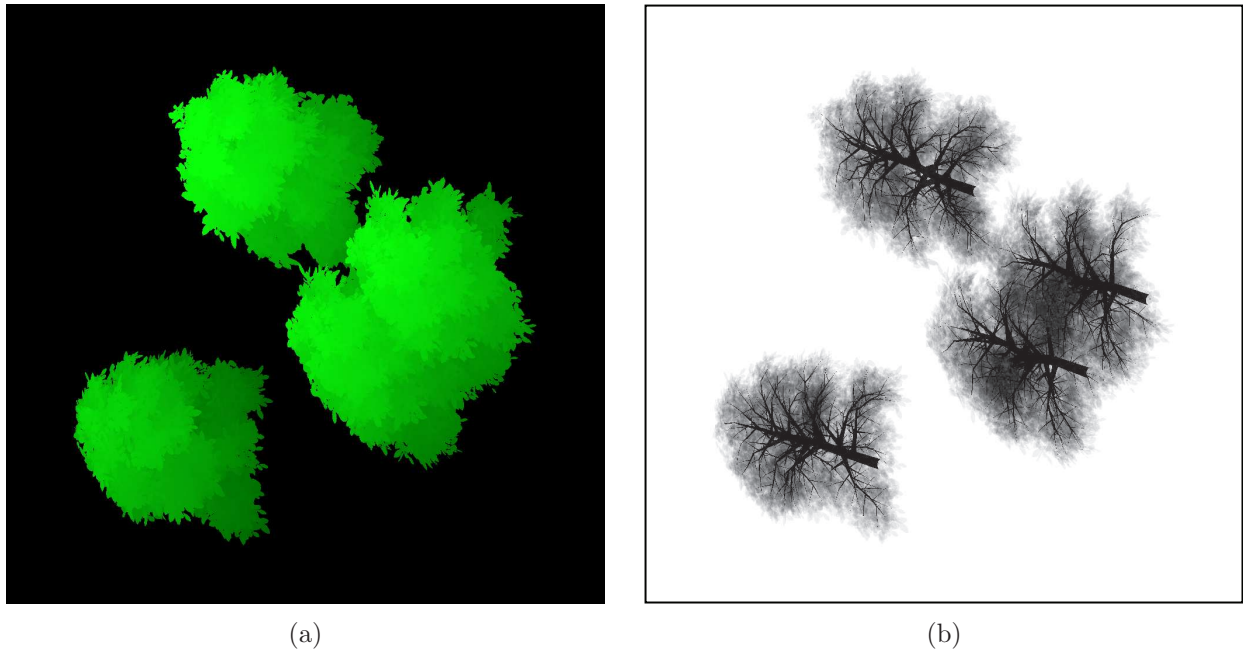


Figure 5.15: (a) RGB channels of the shadow map (only the green channel is used for the  $Y$  axis). (b) Alpha channel of the shadow map (1 = fully transparent, 0 = fully opaque).

### 5.5.2.2 Rendering using the shadow map

At rendering time, each point  $V$  of the objects that can receive shadows (trees and external occluders) is projected onto the shadow map plane at  $P'$  using the same matrix (Equation 5.27) (Figure 5.14(b)). By comparing the length of the resulting  $P'V$  vector to the  $P'P_{max}$  vector stored in the shadow map, we get an algorithm equivalent to traditional shadow mapping. Additionally, we can handle soft shadows. To this end, we use the length of  $VP_{max}$ , distance from the current vertex to the closest occluder to the sun, to compute a blurring factor that simulates soft shadows (Figure 5.23). We bias the mipmap level when accessing the shadow map to get this blurriness. The value we blur is actually the transparency channel

of the shadow map. We want the shadows to look lighter where transparency is higher. An example of fragment shader for shadow projection is given hereafter:

---

**Algorithm 7:** Fragment shader for receiving soft shadows from trees

---

```

varying vec2 shadowMapCoords;
varying vec3 shadowPlaneToReceiver;
uniform sampler2D shadowMap;

float ComputeShadowingFromTrees()
begin
    vec4 shadowMapValue = texture2D (shadowMap, shadowMapCoords, 4.0);
    float blurredThickness = 1.0;
    if shadowMapValue.a < 0.99 then
        float VP = length (shadowPlaneToReceiver * shadowMapValue.y
                            / shadowPlaneToReceiver.y);
        blurredThickness = texture2D (shadowMap, shadowMapCoords,
                                      VP * blurFactor).a;
    end
    return blurredThickness;
end

```

---

A lighter version of the shadowing algorithm is possible. If soft shadows are not desired, the latter fragment shader can be replaced by a single texture access and the alpha channel becomes the result of the `ComputeShadowingFromTrees()` function.

## 5.6 Implementation

Any data structure that can store  $N_{dir}$  values of  $s_{max}$  for each leaf can be used for rendering trees with our model. Our implementation uses a semi-transparent textured quadrilateral per leaf or per small cluster of leaves to render highly detailed trees. Our model results in low frequency lighting hence trees do not flicker when far from the viewer. Therefore it can

also be applied to trees represented as one billboard, a set of billboards or a volume. Each texel or voxel would be assigned the  $s_{max}$  values corresponding to the leaf it represents.

Equations from previous sections seem expensive at first for a GPU implementation. However, many subsets of those equations are constant per tree, hence can be computed on the CPU. The resulting shaders are very light and offer fast rendering of trees in real-time applications. Our lighting model can be evaluated in a vertex shader since most equations are defined per leaf. Only high-frequency shadows and the specular component are evaluated in the fragment shader. Direct lighting due to the sky light and to the sun light is evaluated using Equations 5.6 and 5.8 on the GPU. Only the  $s_{max}(P, \omega_j)$  parameter changes per leaf. Indirect lighting is evaluated on the GPU using Equation 5.24. At rendering time, for each tree,  $G_1$ ,  $G_2$  and  $G_3$  are computed on the CPU using Equations 5.18, 5.25 and 5.26 and can stay constant as long as lighting conditions do not change.

Our implementation uses  $N_{dir} = 8$ . Even though it seems low, it reveals to be enough in practice: Figure 5.16 shows that our results are close to high-quality Monte-Carlo rendering. This low number of samples is very convenient for a GPU implementation. We can store the  $s_{max}$  parameters into two 4D vectors per vertex using texture coordinate sets, which represents only a small memory overhead. Note that the same sets are used when evaluating the three light components, which reduces the overall memory storage. The uniform distribution of sample directions allows us to store only the directions of the first four samples

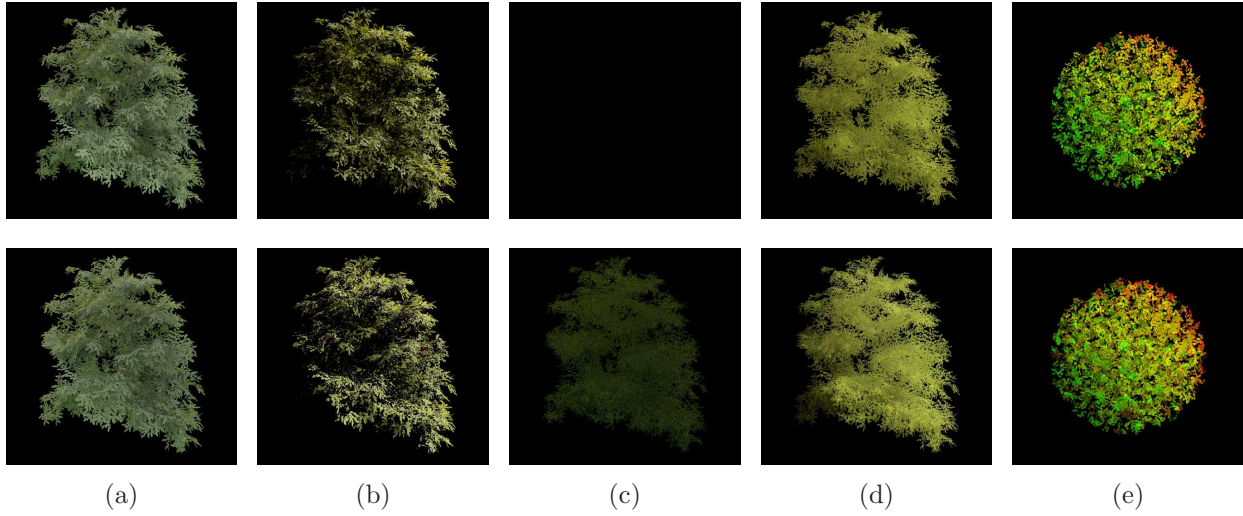


Figure 5.16: Comparing our method to Monte-Carlo rendering. Light comes from the top right corner. Top row: our rendering. Bottom row: Monte-Carlo rendering with 256 rays per intersection point. (a) Direct lighting from the sky. (b) Direct lighting from the sun. (c) Indirect lighting from the sky, ignored in our case. (d) Indirect lighting from the sun. These four light components represent 24%, 48%, 6% and 22% respectively of the total radiance value. (e) Sphere of leaves with false colors for indirect lighting due to the sun, red for reflected light and green for transmitted light.

and recover the others by symmetry as follows:

$$\left\{ \begin{array}{l} \xi = \sqrt{3}/3 \\ \omega_0 = (-\xi, -\xi, -\xi) \\ \omega_1 = (\xi, -\xi, -\xi) \\ \omega_2 = (-\xi, -\xi, \xi) \\ \omega_3 = (\xi, -\xi, \xi) \\ \omega_j = -\omega_{7-j} \quad j \in \{4..7\} \end{array} \right. \quad (5.28)$$



---

**Algorithm 8:** Vertex shader structure for the rendering of leaves

---

```
function LeafVertexShader()
begin
    TransformLeafVertices();           // Algorithm 9
    DetermineLeafMaterials();         // Algorithm 10
    ComputeSkyDirect();               // Algorithm 11
    ComputeSunDirect();               // Algorithm 12
    ComputeSunIndirect();            // Algorithm 13
    FinalizeLeafVertexShader();       // Algorithm 14
end
```

---

---

**Algorithm 9:** Transformation of the leaf vertices and coordinate frame computation

---

```
function TransformLeafVertices()
begin
    // Get the leaf center and the leaf coordinate frame
    vec3 wLeafCenter = GetWorldSpaceLeafCenter();
    vec3 wStaticTangent = GetWorldSpaceLeafTangent();
    vec3 wStaticNormal = GetWorldSpaceLeafNormal();
    vec3 wStaticBinormal = cross(wStaticTangent, wStaticNormal);

    // Rotation of the coordinate frame for animation
    if animated then
        (wTangent, wNormal, wBinormal) =
            RotateFrame(wStaticTangent, wStaticNormal, currentTime);
    else
        (wTangent, wNormal, wBinormal) =
            (wStaticTangent, wStaticNormal, wStaticBinormal);
    end

    // Final position of the vertex
    vec4 wVertex = vec4(wLeafCenter + wTangent * leafWidth
                        + wBinormal * leafHeight, 1);
    vec3 wCameraToVertex = wVertex.xyz - wCameraPos;

    // Position of the vertex in shadow mask space
    vec3 smVertex = [wStaticTangent  wStaticNormal  wStaticBinormal ]T
                    · (wVertex.xyz - wLeafCenter);

    if animated then
        vec3 smLightDir = [wStaticTangent  wStaticNormal  wStaticBinormal ]T ·
            wLightDir;
    end
end
```

---

---

**Algorithm 10:** Computation of material parameters and of the normal pointing to the camera

---

```
function DetermineLeafMaterials()
begin
    // Normal pointing to the camera
    vec3 wNormalTS = ComputeVisibleNormal(wNormal, wLeafCenter- wCameraPos);

    // Visible face and associated reflectance parameters
    float face = dot(wNormal, wNormalTS);
    vec3  $\rho_f$  = max(face, 0) *  $\rho_{r_f}$  + max(-face, 0) *  $\rho_{t_b}$ ;
    vec3  $\rho_b$  = max(face, 0) *  $\rho_{t_f}$  + max(-face, 0) *  $\rho_{r_b}$ ;

    // Projection of the leaf normal to the sampling directions
    vec4 NdotWj03 = vec4(wNormal ·  $\omega_0$ , wNormal ·  $\omega_1$ , wNormal ·  $\omega_2$ , wNormal ·  $\omega_3$ );
    vec8 maxNdotWj = vec8(max(NdotWj03, 0), max(-NdotWj03.wzyx, 0));
    vec8 maxMinusNdotWj = vec8(maxNdotWj [7], ..., maxNdotWj [0]);
end
```

---

---

**Algorithm 11:** Computation of direct lighting due to the sky and its reflection on the ground

---

```
function ComputeSkyDirect()
begin
    // Attenuation of incoming radiance due to leaf layers
    vec8 incomingRed = envIrradianceRed * attenuation;
    vec8 incomingGreen = envIrradianceGreen * attenuation;
    vec8 incomingBlue = envIrradianceBlue * attenuation;

    // Numerical integration
    vec3 skyDirect =  $\rho_f/\pi$  * vec3(dot(maxNdotWj, incomingRed),
                                     dot(maxNdotWj, incomingGreen),
                                     dot(maxNdotWj, incomingBlue))
    +  $\rho_b/\pi$  * vec3(dot(maxMinusNdotWj, incomingRed),
                     dot(maxMinusNdotWj, incomingGreen),
                     dot(maxMinusNdotWj, incomingBlue));
end
```

---

---

**Algorithm 12:** Computation of direct lighting due to the sun

---

```
function ComputeSunDirect()
  begin
    // Estimation of the distance to the tree's border
    float sMaxL = dot(maxLdotWj, sMax) * rcpSumMaxLdotWj;

    // Compute direct lighting with attenuation
    float NTSdotL = dot(wNormalTS, wLightDir);
    vec3 sunDirect = (( $\rho_{r_f} + \rho_{r_b}$ )/ $\pi$  * max(NTSdotL, 0)
                     - ( $\rho_{t_f} + \rho_{t_b}$ )/ $\pi$  * max(-NTSdotL, 0))
                   * lightIntensity * exp(- $\tau$  * sMaxL);
  end
```

---

---

**Algorithm 13:** Computation of indirect lighting due to the sun

---

```
function ComputeSunIndirect()
  begin
    vec8 exp2 = vec8(1) - exp(- $G_3$  * sMax);
    vec8 LoSampleNoCosRed = LoSampleFactorRed * exp2;
    vec8 LoSampleNoCosGreen = LoSampleFactorGreen * exp2;
    vec8 LoSampleNoCosBlue = LoSampleFactorBlue * exp2;

    vec3 sunIndirect =  $\rho_f/\pi$  * vec3(dot(maxNdotWj, LoSampleNoCosRed),
                                       dot(maxNdotWj, LoSampleNoCosGreen),
                                       dot(maxNdotWj, LoSampleNoCosBlue))
                      +  $\rho_b/\pi$  * vec3(dot(maxMinusNdotWj, LoSampleNoCosRed),
                                       dot(maxMinusNdotWj, LoSampleNoCosGreen),
                                       dot(maxMinusNdotWj, LoSampleNoCosBlue));
    sunIndirect *= exp(- $G_2$  * dot(maxLdotWj, sMax)) * leafArea * lightIntensity;
  end
```

---

---

**Algorithm 14:** Final steps of the leaf vertex shader

---

```
function FinalizeLeafVertexShader()
begin
    skyDirectAndSunIndirect = skyDirect + sunIndirect;

    // Distance of the envelope of the tree in [0,1], 1 being the
    // maximum possible distance between a leaf and the tree envelope
    sMaxL /= maxBorderDistance;

    // Texture coordinates
    texCoords = GetVertexTexCoords();

    // Clip space vertex coordinates
    cVertex = projectionMatrix * viewMatrix * wVertex;
end
```

---

The following expressions can be evaluated per tree in the CPU:

$$\begin{aligned} \text{vec8 envIrradiance} &= \frac{\pi}{2} \cdot \text{skyIrradiance}(\omega_j) \\ &\times \begin{cases} 1 & \text{if } \omega_j \cdot y > 0, \\ \text{averageGroundReflectance} & \text{otherwise} \end{cases} \quad j \in \{0..7\} \end{aligned} \quad (5.29)$$

$$\text{vec8 attenuation} = e^{-\tau \cdot \text{sMax}[j]} \quad j \in \{0..7\} \quad (5.30)$$

$$\text{vec8 maxLdotWj} = \max(\omega_l \cdot \omega_j, 0) \quad j \in \{0..7\} \quad (5.31)$$

$$\text{float rcpSumMaxLdotWj} = \frac{1}{\sum_{j=0}^7 \max(\omega_l \cdot \omega_j, 0)} \quad (5.32)$$

$$\text{vec8 LoSampleFactor} = \frac{G_1[j]}{G_3[j]} \quad j \in \{0..7\} \quad (5.33)$$

We now deal with the precomputation of the  $s_{max}$  values. With each leaf centered at  $P$ , we associate a bin with each sampling direction  $\omega_j$ . We then classify each neighbor leaf into one of the bins. Next, we sort the leaves of each bin by increasing order of distance from  $P$ . For each bin, we evaluate the solid angle whose apex is  $P$  and subtended by the closest neighbor leaf. This solid angle divided by the associated bin's solid angle gives an occlusion value in  $]0, 1]$ . By subtracting this value from 1, we get the remaining visibility term. We multiply this term by the occlusion value of the second leaf, compute the new remaining visibility term, and so on for the subsequent leaves. Finally, this results in an attenuation term that asymptotically converges to  $e^{-\tau \cdot s_{max}(P, \omega_j)}$ . We use this term in Equations 5.6 and 5.24. However, the  $s_{max}$  values are necessary in Equation 5.7. Therefore, we need to find the value of  $\tau$ . For each leaf and each associated bin, we first set  $s_{max}$  to the distance of the farthest leaf within the bin and use the attenuation term to compute a value of  $\tau$ . The resulting  $\tau$  values are averaged to get the final value of  $\tau$  for the whole tree. We can then compute the final  $s_{max}$  values for each leaf and each sampling direction using  $\tau$  and the attenuation terms. This precomputation has to be done only once, the resulting attenuation terms are stored in the data structure representing the tree mesh.

## 5.7 Results

Our model can be easily applied to different kinds of tree and to scenes containing many trees (Figure 5.17). It can be easily embedded into existing shaders. Our model handles

dynamic lighting (Figure 5.18) and can be applied to trees animated under the effect of gusts of wind (that modify the leaf normals), as the attenuation terms remain the same as long as the tree branches do not break. Indeed, the lighting updates due to leaf normal changes are fast since they are implemented on the GPU. Unlike our model, PRT methods require the computation of projection coefficients (spherical harmonics, wavelets) for each frame of a sequence of animated trees, which is computationally expensive. Indirect lighting provides an important contribution in situations such as the left image of Figure 5.19 and Figure 5.21 where the sun light goes through the leaves and illuminates neighbor leaves indirectly.

We measured the impact of our lighting model on rendering speed by rendering a set of trees (right image of Figure 5.19) with different sets of light components (see the following table). We tested our algorithm on a *nVidia GeForce 8800 GTX* using HDR lighting. We recall that each rendered tree is highly detailed (about 98,000 triangles).

Light components	1 tree	4 trees	100 trees
direct sky	788 fps	418 fps	36 fps
direct sky + sun	700 fps	349 fps	29 fps
all	667 fps	326 fps	27 fps

Table 5.1: Rendering speed with different light components and different numbers of trees

We remark that the addition of the indirect lighting component has a small impact on performances (less than 7%). As shown in Figure 5.16, our results are close to Monte-Carlo rendering (an average error of 5%, 7.5%, 6% for Figures 5.16(a), 5.16(b) and 5.16(d) respectively). This latter takes 50 minutes to render the image of Figure 5.16 with all light

components combined, while our method only takes 0.4 to 1.5 millisecond per tree depending on the total number of trees.

The precomputation time using a non-optimized implementation is 2 and 40 seconds for trees of 1,500 and 40,000 leaves respectively. This is relatively fast compared to a PRT approach that requires several minutes to several hours of precomputation for a similar amount of vertices [SKS02]. Moreover, our approach is capable of tuning the BSDF while achieving real-time rendering.

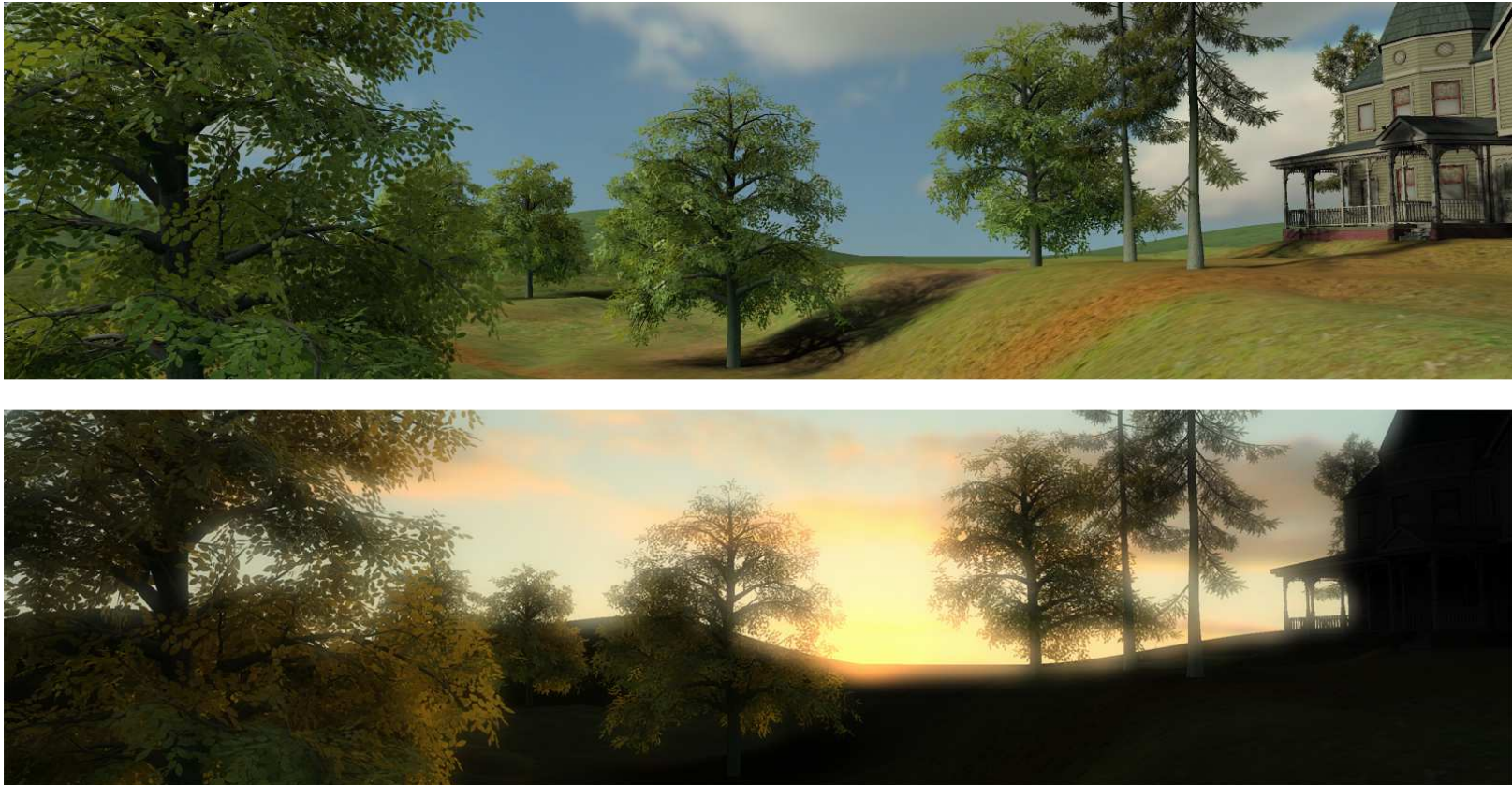


Figure 5.17: The same scene in two different lighting conditions.





Figure 5.18: Our lighting method is dynamic, allowing real-time changes of lighting conditions.



Figure 5.19: The appearance of trees depends mainly on translucency, attenuation and indirect lighting.



Figure 5.20: 100 trees rendered at 27 fps with indirect lighting and shadows.



Figure 5.21: Sunset scene. The left part of the tree has a yellow tint due to the sun color. This yellow tint is scattered inside the tree. Indirect lighting does not reach the right part of the tree where a green color is visible.



Figure 5.22: Shadows cast by leaves onto other leaves.



Figure 5.23: Shadow cast by a tree. Note the sharpness of the shadow close to the trunk and its smoothness farther from the tree.

## CHAPTER 6: CONCLUSION AND FUTURE WORK

Rendering of nature scenes in real-time has always been a difficult problem due to the required computing power for geometry and lighting processing. We have been able to reach this goal to a certain extent: our grass rendering algorithm allows a virtually infinite number of grass blades to be rendered in real-time and our tree global illumination approach allows hundreds of procedural trees of thousands leaves each to be rendered in real-time. However, we do not want to stop there, many open problems are still to be solved. Speed is crucial in real-time applications, but visual quality is also very important. Lighting has to be as accurate as possible while having a low impact on performance. Approximations that give the most visually convincing results at the lowest possible cost are required.

The objective of our work on grass was to render dynamically lit, shadowed, anti-aliased grass with variable density on large terrains in real-time. We achieved this goal by mixing geometry and volume rendering. Grass modeled with geometry is used for rendering of grass near the viewer, volume rendering is used at moderate distances and only the horizontal slice of volume rendering is used for distant grass. For the volume rendering algorithm, we discretize a volume representing the base grass patch into slices aligned with the world axes for a good parallax effect. For each slice we make use of textures defined for five sample light directions taking self-shadowing and self-occlusions into account. We use spherical

barycentric interpolation to compute the reflected luminance for the inbetween incident light directions. Besides our contribution of handling dynamic lighting and shadows, we introduced the notion of density to create arbitrarily shaped areas of grass. This density also allows dynamic management of seamless transitions between two levels of detail depending on the position and direction of the viewer. Additionally, our approach also handles curved terrains with a very low additional cost.

This work is extensible to any type of furry surface with variable density and to other natural elements such as trees, plants, flowers, etc. Our method currently uses diffuse only reflectance model for grass surfaces. One direction of future research would be to replace the diffuse reflectance by a more complex reflectance model.

The goal of our work on trees was to overcome the lighting complexity without defining an empirical model based only on observations of real trees. To this end, we presented an approach based on the fact that human vision is more sensitive to the overall aspect of objects rather than precise details and we designed a lighting model for trees that allows real-time rendering with indirect illumination. We approximated the scattering of light inside sets of leaves by giving them probabilistic properties, and thus derived an analytical solution for low-frequency lighting. We achieved high speedups by reducing the computation load on parameters that have the least influence on the appearance. The implementation of our model on the GPU is straightforward. Furthermore, many parameters can be quickly computed per tree instance on the CPU, allowing short and efficient shaders.



Extending our model to point light sources can be a subject of future work. This would allow to render trees close to street lamps or trees on fire. Another extension is to account for the illumination of the ground and trees by other trees through reflection and transmission. Our approach makes use of uniform distribution of leaf positions and normals. Using non-uniform distributions could increase the accuracy of the computations and can be subject to future work. A future project could integrate our tree lighting algorithm into an image-based approach. When trees are rendered close to the viewer, many fragments are generated for a relatively low number of processed vertices. That is why our algorithm is executed per vertex. For farther trees, the number of pixels becomes low, sometimes so low that it is less than the number of vertices processed for each tree. In this case, it will be beneficial to use an image-based approach: the lighting algorithm is executed per fragment and only a few vertices are processed per tree (just enough to model the billboards). That should allow us to get many trees rendered at low cost, and the possibility to obtain globally illuminated forests in real-time.

## **APPENDIX A: SPHERICAL BARYCENTRIC INTERPOLATION OF THE GRASS BTF DATA**

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 when moving the camera, hence the need for interpolation. The parameters we define are presented in Figure A.1. The vector  $l = OL$  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 follows:

$$C_{int} = \alpha_x C_{X+} + \alpha_y C_Y + \alpha_z C_{Z+} \quad (\text{A.1})$$

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

$$C_{int} = \frac{\mathcal{A}(LBC)}{\mathcal{A}(ABC)} C_{X+} + \frac{\mathcal{A}(LCA)}{\mathcal{A}(ABC)} C_Y + \frac{\mathcal{A}(LAB)}{\mathcal{A}(ABC)} C_{Z+} \quad (\text{A.2})$$



This expression can be simplified using the following relations:

$$\left\{ \begin{array}{l} \cos \theta_y = l \cdot y = l_y \\ \cos \theta_z = l \cdot z = l_z \\ \sin \theta_y = \|l \times y\| = \|(-l_z, 0, l_x)^T\| = \sqrt{l_z^2 + l_x^2} \\ \sin \theta_z = \|l \times z\| = \|(l_y, -l_x, 0)^T\| = \sqrt{l_x^2 + l_y^2} \end{array} \right. \quad (\text{A.5})$$

To find the angles of Equation A.3, we take the arccosines of Equations A.4:

$$\begin{aligned} \mathcal{A}(LBC) = & \arccos \frac{l_z}{\sqrt{l_z^2 + l_x^2}} + \arccos \frac{l_y}{\sqrt{l_x^2 + l_y^2}} \\ & + \arccos \frac{-l_y l_z}{\sqrt{l_z^2 + l_x^2} \sqrt{l_x^2 + l_y^2}} - \pi \end{aligned} \quad (\text{A.6})$$

We apply the same process for the areas  $\mathcal{A}(BCL)$  and  $\mathcal{A}(CLB)$ . Finally, we obtain the surfaces of the three spherical triangles, and are able to compute  $\alpha_x$ ,  $\alpha_y$  and  $\alpha_z$  of Equation A.1.

These coefficients are computed per vertex rather than per pixel to greatly reduce the overhead at the GPU level. This approximation is valid if the slices length is small compared to the distance from the light source to the slices. 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. This looks intensive for the GPU, however the bottleneck of slices rendering is the fragment shading. So the overhead 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.

**APPENDIX B: SIMPLIFICATION OF EQUATION  
5.15**

The evaluation of the outgoing radiance  $L_{O_{slice}}$  from all neighbor leaves centered at  $P'$  inside the slice  $V_{j,i}$  requires many expensive sums (Equation 5.15). However, we want a computationally inexpensive analytical model for  $L_{O_{slice}}$ . The four sums have similar expressions, hence we will calculate the second one and find the others by symmetry. The number of leaves  $N_{leaves_{j,i}}$  in a slice  $V_{j,i}$  is considered as high and the distribution of normals  $N_{P'}$  is uniform. Therefore, the sums are close to their corresponding integrals as follows:

$$\begin{aligned}
& \sum_{P' \in V_{j,i}} M(N_{P'} \cdot \omega_l) \cdot M(N_{P'} \cdot \omega_j) \\
& \approx \frac{N_{leaves_{j,i}}}{4\pi} \int_{\Omega} M(N_{P'} \cdot \omega_l) \cdot M(N_{P'} \cdot \omega_j) dN_{P'} \\
& = N_{leaves_{j,i}} \cdot Av_t(\omega_l, \omega_j)
\end{aligned} \tag{B.1}$$

with  $Av_t(\omega_l, \omega_j)$  the average of  $M(N_{P'} \cdot \omega_l) \cdot M(N_{P'} \cdot \omega_j)$  given  $\omega_l$  and  $\omega_j$ .

$$Av_t(\omega_l, \omega_j) = \frac{1}{4\pi} \int_{\Omega} H(N_{P'} \cdot \omega_l) \cdot (N_{P'} \cdot \omega_l) \cdot H(N_{P'} \cdot \omega_j) \cdot (N_{P'} \cdot \omega_j) dN_{P'} \tag{B.2}$$

where  $H(x)$  is the Heaviside step function, equal to 1 if  $x \geq 0$ , 0 otherwise. The uniform distribution of  $N_{P'}$  makes  $Av_t(\omega_l, \omega_j)$  depending only on the angle  $\alpha_j$  between  $\omega_l$  and  $\omega_j$ .

We now express the average as  $Av_t(\alpha_j)$ . As shown in Figure B.1, we define  $\omega_j = X$  and  $\omega_l$  in the  $(O, X, Z)$  plane with an angle  $\alpha_j$  with  $\omega_j$ . We express  $N_{P'} = (\varphi, \theta)$  with spherical



coordinates and obtain the following expression for  $Av_t(\alpha_j)$ :

$$Av_t(\alpha_j) = \frac{1}{4\pi} \int_{\varphi=0}^{2\pi} \int_{\theta=0}^{\pi} H(\sin \theta(\cos \alpha_j \cos \varphi + \sin \alpha_j \sin \varphi)) \cdot (\sin \theta(\cos \alpha_j \cos \varphi + \sin \alpha_j \sin \varphi)) \cdot H(\cos \varphi \sin \theta) \cdot (\cos \varphi \sin \theta) \cdot \sin \theta \, d\theta d\varphi \quad (\text{B.3})$$

To find an analytical form to this integral, we first need to remove the step functions by modifying the bounds of the integrals. With  $\theta \in ]0, \pi[$ ,  $\sin \theta > 0$ . We also have  $\cos \varphi > 0$  if  $\varphi \in ]0, \pi/2[ \cup ]3\pi/2, 2\pi[$ . With these properties, the second step function can be removed and the integrals split into two parts. To remove the first step function, we need two cases:  $\alpha_j \in ]0, \pi/2[$  and  $\alpha_j \in ]\pi/2, \pi[$ . In the first case, we have  $\cos \alpha_j > 0$  and  $\sin \alpha_j > 0$ . If  $\varphi \in ]0, \pi/2[$ , the first step function is always equal to 1. If  $\varphi \in ]3\pi/2, 2\pi[$ , the first step function is equal to 1 only if  $\varphi > 3\pi/2 + \alpha_j$ . We then get

$$\begin{aligned} Av_t(\alpha_j) &= \frac{\cos \alpha_j}{4\pi} \int_0^{\frac{\pi}{2}} \int_0^{\pi} \cos^2 \varphi \sin^3 \theta \, d\theta d\varphi \\ &+ \frac{\cos \alpha_j}{4\pi} \int_{\frac{3\pi}{2} + \alpha_j}^{2\pi} \int_0^{\pi} \cos^2 \varphi \sin^3 \theta \, d\theta d\varphi \\ &+ \frac{\sin \alpha_j}{4\pi} \int_0^{\frac{\pi}{2}} \int_0^{\pi} \cos \varphi \sin \varphi \sin^3 \theta \, d\theta d\varphi \\ &+ \frac{\sin \alpha_j}{4\pi} \int_{\frac{3\pi}{2} + \alpha_j}^{2\pi} \int_0^{\pi} \cos \varphi \sin \varphi \sin^3 \theta \, d\theta d\varphi \end{aligned} \quad (\text{B.4})$$

Solving this equation gives the expression of  $Av_t(\alpha_j)$  for  $\alpha_j \in ]0, \pi/2[$ :

$$Av_t(\alpha_j) = \frac{1}{6\pi} ((\pi - \alpha_j) \cos \alpha_j + \sin \alpha_j) \quad (\text{B.5})$$

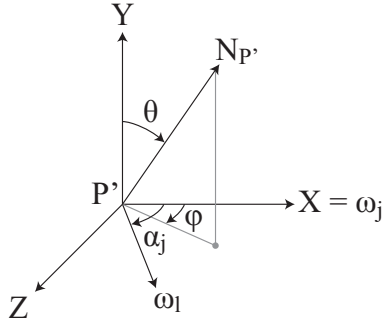


Figure B.1:  $N_{P'}$ ,  $\omega_j$  and  $\omega_l$  in spherical coordinates.

The  $\alpha_j \in ]\pi/2, \pi[$  case can be calculated in a similar way and actually gives exactly the same result as Equation B.5. We prolong by continuity to have  $Av_t(\alpha_j)$  for  $\alpha_j \in [0, \pi]$ .

Due to the uniform distribution of  $N_{P'}$ , the number of leaves for which light goes through from the front face to the back face is the same as the number of leaves for which light goes through from the back face to the front face. Therefore, the third sum of Equation 5.15 can be expressed exactly as the second sum using Equations B.1 and B.5.

To get the expressions of the first and fourth sums, we apply a symmetry to the problem and use  $\alpha'_j = \pi - \alpha_j$ . We use  $\alpha'_j$  in place of  $\alpha_j$  for all previous equations of this section. We finally get  $Av_r$  that replaces  $Av_t$  in Equation B.1:

$$Av_r(\alpha_j) = \frac{1}{6\pi} (-\alpha_j \cos \alpha_j + \sin \alpha_j) \quad \alpha_j \in [0, \pi] \quad (\text{B.6})$$

## LIST OF REFERENCES

- [AH02] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering - Second Edition*, chapter 12.1.3, pp. 492–494. A K Peters, 2002.
- [AMB07] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. “Convolution Shadow Maps.” In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007: Eurographics Symposium on Rendering*, volume 18, pp. 51–60. Eurographics, June 2007.
- [AMS08] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. “Exponential Shadow Maps.” In *Proceedings of Graphics Interface*, May 2008.
- [BLH02] Brook Bakay, Paul Lalonde, and Wolfgang Heidrich. “Real Time Animated Grass.” In *Eurographics 2002*, 2002.
- [Bli77] James F. Blinn. “Models of light reflection for computer synthesized pictures.” In *SIGGRAPH ’77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pp. 192–198, New York, NY, USA, 1977. ACM Press.
- [BM02] Jitendra A. Borse and David F. McAllister. “Real-Time Image-Based Rendering for Stereo Views of Vegetation.” In *Proceedings of Stereoscopic displays and virtual reality systems IX*, volume 4660, pp. 292–299, San Jose, CA, January 2002.
- [CAB98] Michal Chelle, Bruno Andrieu, and Kadi Bouatouch. “Nested radiosity for plant canopies.” *The Visual Computer*, **14**(3):109–125, 1998.
- [CSH03] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. “Wang Tiles for Image and Texture Generation.” *ACM Transactions On Graphics*, **22**(3):287–294, 2003.
- [DCD05] Andreas Dietrich, Carsten Colditz, Oliver Deussen, and Philipp Slusallek. “Realistic and Interactive Visualization of High-Density Plant Ecosystems.” In *Natural Phenomena 2005, Proceedings of the Eurographics Workshop on Natural Phenomena*, pp. 73–81, August 2005.

- [DCS02] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. “Interactive visualization of complex plant ecosystems.” In *Proceedings of the IEEE Visualization Conference*. IEEE, October 2002.
- [DHL98] 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*, pp. 275–286. ACM, 1998.
- [DMS06] Andreas Dietrich, Gerd Marmitt, and Philipp Slusallek. “Terrain Guided Multi-Level Instancing of Highly Complex Plant Populations.” In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pp. 169–176, September 2006.
- [DN04] Philippe Decaudin and Fabrice Neyret. “Rendering Forest Scenes in Real-Time.” In A. Keller H. W. Jensen, editor, *Eurographics Symposium on Rendering*, 2004.
- [EHK06] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, and Daniel Weiskopf. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.
- [GMW04] Callum Galbraith, Lars Mundermann, and Brian Wyvill. “Implicit Visualization and Inverse Modeling of Growing Trees.” *Computer Graphics Forum*, **23**(3):351–360, 2004.
- [GPR03] Sylvain Guerraz, Frank Perbet, David Raulo, François Faure, and Marie-Paule Cani. “A Procedural Approach to Animate Interactive Natural Sceneries.” In *CASA '03: Proceedings of the 16th International Conference on Computer Animation and Social Agents (CASA 2003)*, p. 73, Washington, DC, USA, 2003. IEEE Computer Society.
- [Hei91] Tim Heidmann. “Real Shadows Real Time.” *IRIS Universe*, (18):28–31, November 1991.
- [HKW07] Ralf Habel, Alexander Kusternig, and Michael Wimmer. “Physically Based Real-Time Translucency for Leaves.” In *Rendering Techniques (Proceedings of EGSR)*, pp. 253–263, June 2007.
- [HPA06] Kyle Hegeman, Simon Premoze, Michael Ashikhmin, and George Drettakis. “Approximate Ambient Occlusion For Trees.” In *Proceedings of the ACM SIG-GRAPH Symposium on Interactive 3D Graphics and Games*, March 2006.
- [IKL04] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. *GPU Gems*, chapter 39 - Volume Rendering Techniques, pp. 667–692. Addison-Wesley, March 2004.
- [Jak00] Aleks Jakulin. “Interactive Vegetation Rendering with Slicing and Blending.” In *Proceedings of Eurographics 2000 (Short Presentations)*, August 2000.

- [KK89] James T. Kajiya and Timothy L. Kay. “Rendering fur with three dimensional textures.” In *Computer Graphics (SIGGRAPH 89 Proceedings)*, volume 23(3), pp. 271–280. ACM, July 1989.
- [KL05] Janne Kontkanen and Samuli Laine. “Ambient occlusion fields.” In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pp. 41–48, New York, NY, USA, 2005. ACM Press.
- [KPH03] Joe Kniss, Simon Premoze, Charles Hansen, Peter Shirley, and Allen McPherson. “A Model for Volume Lighting and Modeling.” In *IEEE Transactions on Visualization and Computer Graphics*, volume 9(2), pp. 150–162, 2003.
- [LBD07] Thomas Luft, Michael Balzer, and Oliver Deussen. “Expressive Illumination of Foliage Based on Implicit Surfaces.” In *Eurographics Workshop on Natural Phenomena, 2007*.
- [Lew94] Robert R. Lewis. “Making Shaders More Physically Plausible.” *Computer Graphics Forum (Eurographics '94 Conference Issue)*, **13**(3):1–13, 1994.
- [LFT97] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. “Non-linear approximation of reflectance functions.” In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 117–126. ACM Press/Addison-Wesley Publishing Co., 1997.
- [Lin68] Aristid Lindenmayer. “Mathematical models for cellular interactions in development, parts I and II.” *Journal of Theoretical Biology*, **18**(3):280–315, March 1968.
- [LP02] Brendan Lane and Przemyslaw Prusinkiewicz. “Generating Spatial Distributions for Multilevel Models of Plant Communities.” In *Proceedings of Graphics Interface*, pp. 69–80, May 2002.
- [LPF01] Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. “Real-Time Fur over Arbitrary Surfaces.” In *ACM Symposium on Interactive 3D Graphics*, pp. 227–232. ACM, March 2001.
- [Max96] Nelson Max. “Hierarchical rendering of trees from Precomputed Multi-Layer Z-Buffers.” In X. Pueyo and P. Schroeder, editors, *Rendering Techniques, '96 (Proceedings of 7th Eurographics Workshop on Rendering)*, New York, 1996. Springer.
- [McA04] David McAllister. *GPU Gems*, chapter 18 - Spatial BRDFs, pp. 293–306. Addison-Wesley, March 2004.
- [McC94] Ross McCluney. *Introduction to Radiometry and Photometry*. Artech House, Inc., Norwood, MA, USA, 1994.

- [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.
- [MMS04] Gero Müller, Jan Meseth, Mirko Sattler, Ralf Sarlette, and Reinhard Klein. “Acquisition, Synthesis and Rendering of Bidirectional Texture Functions.” In Christophe Schlick and Werner Purgathofer, editors, *Eurographics 2004, State of the Art Reports*, pp. 69–94. INRIA and Eurographics Association, September 2004.
- [MN98] Alexandre Meyer and Fabrice Neyret. “Interactive Volumetric Textures.” In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop 1998*, pp. 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*, pp. 183–196, July 2001.
- [MO95] Nelson Max and Keiichi Ohsaki. “Rendering Trees from Precomputed Z-Buffer Views.” In *Proceedings of the 6th Eurographics Workshop on Rendering*, 1995.
- [Ney95a] Fabrice Neyret. “Animated Texels.” In *Eurographics Workshop on Animation and Simulation 95*, pp. 97–103, September 1995.
- [Ney95b] Fabrice Neyret. “A General and Multiscale Model for Volumetric Textures.” In *Proceedings of Graphics Interface 95*, pp. 83–91, May 1995.
- [Ney96] Fabrice Neyret. “Synthesizing Verdant Landscapes using Volumetric Textures.” In X. Pueyo and P. Schroöder, editors, *Rendering Techniques 96*, pp. 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.
- [ON94] Michael Oren and Shree K. Nayar. “Generalization of Lambert’s reflectance model.” In *SIGGRAPH ’94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 239–246, New York, NY, USA, 1994. ACM Press.
- [OOI05] Makoto Okabe, Shigeru Owada, and Takeo Igarash. “Interactive Design of Botanical Trees using Freehand Sketches and Example-based Editing.” *Computer Graphics Forum*, **24**(3):487–496, 2005.
- [Pat93] Chris Patmore. “Illumination in Dense Foliage Models.” In *Proceedings of the Fourth Eurographics Workshop on Rendering*, pp. 63–71, 1993.

- [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*. ACM, Eurographics, ACM Press, 2001.
- [Pel04] Kurt Pelzer. *GPU Gems*, chapter 7 - Rendering Countless Blades of Waving Grass, pp. 107–121. Addison-Wesley, March 2004.
- [Pho75] Bui Tuong Phong. “Illumination for computer generated pictures.” *Communications of the ACM*, **18**(6):311–317, 1975.
- [PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.
- [PL06] Scott Peterson and Lawrence Lee. “Simplified Tree Lighting using Aggregate Normals.” In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, p. 47, 2006.
- [QNT03] Xueying Qin, Eihachiro Nakamae, Katsumi Tadamura, and Yasuo Nagai. “Fast Photo-Realistic Rendering of Trees in Daylight.” *Computer Graphics Forum*, **22**(3):243–252, 2003.
- [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), pp. 313–322. ACM, July 1985.
- [Ree83] William T. Reeves. “Particle Systems – A Technique for Modeling a Class of Fuzzy Objects.”, July 1983.
- [RH01] Ravi Ramamoorthi and Pat Hanrahan. “An efficient representation for irradiance environment maps.” In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 497–500, New York, NY, USA, 2001. ACM Press.
- [RMD04] Alex Reche, Ignacio Martin, and George Drettakis. “Volumetric Reconstruction and Interactive Rendering of Trees from Photographs.” *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)*, **23**(3), July 2004.
- [Sch94] Christophe Schlick. “An Inexpensive BRDF Model for Physically-Based Rendering.” *Computer Graphics Forum*, **13**(3):233–246, 1994.
- [SD02] Marc Stamminger and George Drettakis. “Perspective Shadow Maps.” In *Proceedings of ACM SIGGRAPH 2002*, July 2002.
- [SKP05] Musawir A. Shah, Jaakko Konttinen, and Sumanta Pattanaik. “Real-time rendering of realistic-looking grass.” In *Proceedings of GRAPHITE '05*, pp. 77–82, November 2005.

- [SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. “Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments.” In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 527–536, New York, NY, USA, 2002. ACM.
- [SSB03] Cyril Soler, François Sillion, Frédéric Blaise, and Philippe Dereffye. “An Efficient Instantiation Algorithm for Simulating Radiant Energy Transfer in Plant Models.” *ACM Transactions On Graphics*, **22**(2), April 2003.
- [TS67] Kenneth E. Torrance and E.M. Sparrow. “Theory for Off-Specular Reflection from Roughened Surfaces.” *Journal of Optical Society of America*, **57**:1105–1114, 1967.
- [Wha05] David Whatley. *GPU Gems 2*, chapter 1 - Toward Photorealism in Virtual Botany, pp. 7–25. Addison-Wesley, March 2005.
- [Wil78] Lance Williams. “Casting curved shadows on curved surfaces.” In *Proceedings of SIGGRAPH 78*, pp. 270–274, 1978.
- [WP95] Jason Weber and Joseph Penn. “Creation and rendering of realistic trees.” In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 119–128, New York, NY, USA, 1995. ACM Press.
- [WWD05] Lifeng Wang, Wenle Wang, Julie Dorsey, Xu Yang, Baining Guo, and Heung-Yeung Shum. “Real-time Rendering of Plant Leaves.” In *ACM Transactions On Graphics (Proceedings of SIGGRAPH 2005)*, pp. 712–719, 2005.
- [XFr] Greenworks organic software XFrog. “<http://www.xfrog.com>.”.