Annexes

articles publiés

A Représentations géométriques alternatives

- *Interactive Volumetric Textures* (EWR'98)

- *Multiscale Shaders for the Efficient Realistic Rendering of Pine-Trees* (GI'00)

- *Interactive Rendering of Trees with Shading and Shadows* (EWR'01)

Interactive Volumetric Textures

Alexandre Meyer and Fabrice Neyret

iMAGIS*, laboratoire GRAVIR/IMAG-INRIA
{Alexandre.Meyer|Fabrice.Neyret}@imag.fr
http://w3imagis.imag.fr/Membres/Fabrice.Neyret/index.gb.html

Abstract:
This paper presents a method for interactively rendering complex repetitive scenes such as landscapes, fur, organic tissues, etc. It is an adaptation to Z-buffer of *volumetric textures*, a ray-traced method, in order to use the power of existing graphics hardware. Our approach consists in slicing a piece of 3D geometry (one repetitive detail of the complex data) into a series of thin layers. A layer is a rectangle containing the shaded geometry that falls in that slice. These layers are used as transparent textures, that are mapped onto the underlying surface (e.g. a hill or an animal skin) with an extrusion offset. We show some results obtained with our first implementation, such as a scene of 13 millions of virtual polygons animated at 2.5 frames per second on a SGI O_2.

1 Introduction

Visual complexity is part of the realism of a scene, especially for natural scenes like landscapes, fur, organic tissues, etc. When represented explicitly with facets, these complex -and often repetitive- details lead to very high rendering time and aliasing artifacts. In some cases these details are flat enough to be represented with flat textures. However in many case they are really three-dimensional, i.e. showing view-dependent appearance and parallax motion (e.g. trees on a hill). Moreover, mesh decimation algorithms are of no help on such complex objects. The situation is even worse in the scope of interactive rendering, where only very low complexity scenes can usually be dealt with in the available time.

The fact that a detail is not flat does not imply it has to be represented by a comprehensive - and costly - 3D representation such as a mesh. Indeed, the 3D impression is a progressive notion: it includes several properties, such as view-dependent contour, view-dependent apparent location, parallax motion, occlusion, shadowing, diffuse reflection and highlights, etc. Depending on the size of the object (or the detail) on the screen, some of these properties can be sufficient to convey a 3D impression. A means to do efficient rendering with few aliasing artifacts is thus to use a representation that refers to the minimum amount of information that is sufficient to reproduce what can be seen.

1.1 Related work

Volumetric textures, introduced in 1989 by Kajiya and Kay [4] and extended by us [8, 9], consider three different embedded scales to represent the information (see figure 1):

- large shape variations such as the surface of a hill or an animal skin are encoded using a regular surface mesh,
- the medium scale such as grass or skin, which is concentrated in the neighborhood of this surface, is encoded using a *reference volume* stored once and mapped several times in the spirit of textures (instances are named *texels*),

* *iMAGIS* is a joint research project of CNRS/INRIA/UJF/INPG.
Postal address: BP 53, F-38041 Grenoble cedex 09, France.

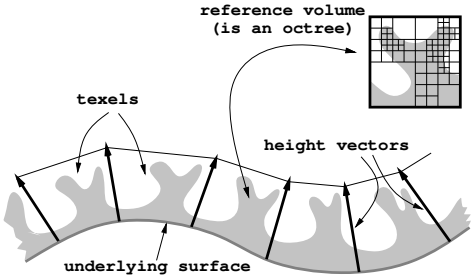


Fig. 1. Volumetric texture specification (cross-section).

- the small scale, consisting of the microscopic shape of individual objects, is encoded by a reflection model stored in each voxel. In the multiscale extension of volumetric textures, this scale also corresponds to the pixel size.

To relate this to the progressive 3D impression mentioned before, one can see that explicit geometry is used only to specify the largest scale; volume data is sufficient[1] to reproduce occlusions and parallax effects at middle scale (i.e. a few pixels), while the illumination model stored in the voxels simulates the geometry below pixel size.

Texel rendering has some similarities with volume rendering, a previously costly ray-tracing method family. In 1994 Lacroute and Levoy introduced a new approach [5] adapted to graphics hardware, which makes volume rendering interactive. This approach consists of factoring the voxels by considering slices of the volume, that can be encoded by textured transparent faces. Volume rendering thus consists of superimposing these transparent slices. Since common 3D graphics hardware can deal with textures and transparency at no extra cost, rendering cost is now only proportional to the number of slices.

1.2 Overview

This paper presents a method for interactive rendering of complex repetitive geometry. The idea is to adapt the volumetric textures presented above [8, 9] to Z-buffer graphics hardware, using the same sort of approach that was used for volume rendering by Lacroute and Levoy [5]. We thus expect to obtain the same kind of complex scene as the first, with the same kind of interactiveness as the second.

Contrary to volume rendering the size of the volumes used in our method is small. We can thus render a relatively large number of such volumes interactively. For instance, a volume encoded with 64 slices can be rendered with a cost of 64 quadrilateral faces (i.e. 64×2 triangles), while the represented shape is built from a model that might have at least a thousand faces (and often ten or a hundred times more). Moreover, this cost is independent of the slice resolution.

On the other hand, using graphics hardware brings some limitations: with hardware rendering one cannot compute shading nor shadows for each individual texture pixel (i.e. for each volume cell), while ray-tracing can do this. Only color is stored in the volume, so the shading and shadows - if any - have to be captured inside the pattern at the creation stage, and will not be updated according to the main surface orientation and light position.

[1] Because of the concentration of the data complexity within the surface neighborhood, and the small volume resolution necessary to provide 3D location effect, in our context a volume is an efficient and compact way to store and render data.

The remainder of this paper is structured as follows. In section 2, we deal with the basic representation and rendering of interactive volumetric textures, that will be extended in section 4. In section 3 we describe how to encode the shape of a detail into a texel, in particular by converting existing representations. Animation approaches available for the ray-tracing method [7] are still usable for ours. We review these approaches in section 5. We discuss the results in section 6.

2 Basic representation and rendering

In this section, we present our method to encode a complex object made of repetitive details lying on a surface, and explain how to render the representation obtained. The modeling of the content is the object of the next section.

2.1 Data structure

In the same way as ray-traced volumetric textures [8], the specification of an object consists of a triangular mesh with (u, v) texture coordinates and a height vector at the vertices, plus a volumetric texture pattern. The height vectors control the direction and thickness of the third dimension of the texture (see figure 1).

The volumetric part of the model is different to the one used for the ray-traced version: it consists of a set of RGBA textures, representing infinitely thin horizontal slices of the volume. Empty parts have $A = 0$ (i.e. the slice is transparent there), and opaque parts have $A = 255$.

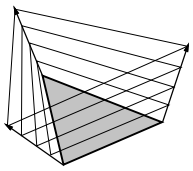


Fig. 2. A texel is drawn using extruded textured triangles.

2.2 Rendering

The rendering is done using a standard hardware-accelerated 3D graphics library (OpenGL [6], in our implementation), by drawing textured extruded facets above each geometric facet of a "volumetrically textured" surface. The three vertices of an extruded facet (corresponding to a slice) are obtained by linearly interpolating the position along the three height vectors at the three vertices of the surface facet (as illustrated in figure 2). Hardware MIP-mapping [13] can be used to deal with aliasing at grazing view angles and distant location. Notice that texture, transparency and MIP-mapping come at no extra rendering cost[2] on various 3D graphics cards.

It is known that transparency does not work well with a Z-buffer; correct transparency would require storing several Z, alpha and color values per pixel. As long as the alpha value A is 0 or 255, this is not a problem: transparent texture pixels are not drawn, and opaque texture pixels hide what is behind them. However a problem occurs when semi-transparent texture pixels exist, either because the content is smoothed or the MIP-mapping feature is on. To deal correctly with this problem, one has to draw the slices from back to front. This is easy to achieve within a single texel, but this would also require to sort the faces with Z, which is costly. Thus, we do not allow semi-transparent data in our implementation. However we do draw the slices from back to front, since this avoid the artifacts that may occur due to the lack of resolution in Z between slices.

[2] To a certain extent, beyond which hardware bottlenecks occur.

To choose the drawing order, it is sufficient to test the dot product of the normal to the surface facet and the view direction, assuming that the texel is not too distorted by the height vectors.

Each volume location is treated by the Z-buffer as a regular pixel fragment (i.e. it has its own Z-value), thus the intersection of two texels is dealt with correctly, which was not the case in the ray-tracing version. This important property is illustrated in figure 10(right) in the results section 6.

3 Modeling the pattern

In this section we describe how to encode in a texel one repetitive detail of a complex object. This detail is created using an existing modeling tool. However, using a textural approach to repeat the detail brings some constraints to the pattern shape: the 3D texture pattern, i.e. the reference volume of the volumetric texture, corresponds to the cubic box between $(u, v) = (0, 0)$ and $(u, v) = (1, 1)$ in the texture space. The mapping will

Fig. 3. *Left*: Pattern with torus topology. *Right*: Isolated pattern.

generate the (virtual) copies of the detail according to the (u, v) texture coordinates at the vertices. For the result of the mapping to appear continuous, the cubic pattern content has either to obey torus topology, or to consist of a disconnected shape that does not reach the borders of the reference volume, as shown in figure 3.

Once an appropriate shape has been chosen or modeled for the detail, it has to be encoded in the volume (i.e. the set of texture slices) in such a way that the texel reproduces the same visual effect. This leads to several stages in the encoding:

- slicing the 3D description,
- evaluating the shading at each location,
- filling the inside of the shape.

The last issue is a key point: if the description is a surface, and not a solid, each slice of it is a contour (see figure 4(left)), so gaps would appear between slices when the view direction is not orthogonal to the surface. Thus the shape has to be solid, or to be turned into solid if the description is a surface[3]. Some inside slice pixels are visible between two contours[4] as shown in figure 4(middle). We need to propagate the surface color toward the inside, in such a way that the image appears as continuous as possible.

This approach is not sufficient for grazing view angles, because the gap between slices appears, as illustrated in figure 4(right). This problem is solved in section 4.

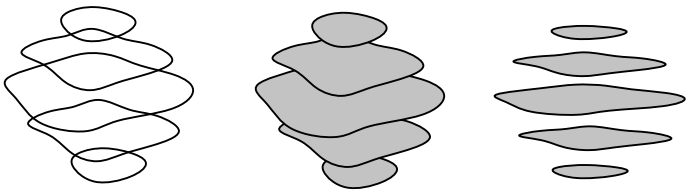


Fig. 4. *Left*: Contours. *Middle*: Filled contours. *Right*: Gaps appearing between slices at grazing view angles.

[3] Of course, this does not concern shapes made of sparse polygons, e.g. foliage.

[4] Here, we only deal with opaque solids.

3.1 Slicing and shading the pattern shape

In the case of a standard surface description (e.g. an OpenInventor database), one can use a standard renderer (e.g. OpenGL) to do the slicing and shading at the same time. The view point is set at the top of the 3D pattern, a bounding box is defined by the user, then the front and back clipping planes are successively set around each slice (as illustrated in figure 5). Each resulting RGBA image is stored as a texture slice (including the alpha value, which is crucial), and the slices set is stored on disk.

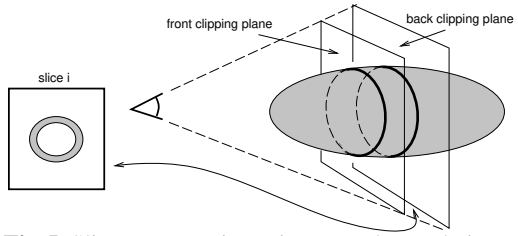


Fig. 5. Slices construction using a regular rendering tool.

A ray-tracer may be used as well for the rendering of the slices, which would allow for shadows. However, as for the shading, one has to keep in mind that the considered light directions will be fixed at this construction stage.

3.2 Filling the inside

For surface descriptions, the slices are empty contours, that need to be filled. Worse, these contours are incomplete. Thus the filling comprises three stages:

- closing the contours,
- marking the inside (i.e. where to propagate the color),
- performing the color propagation within a slice.

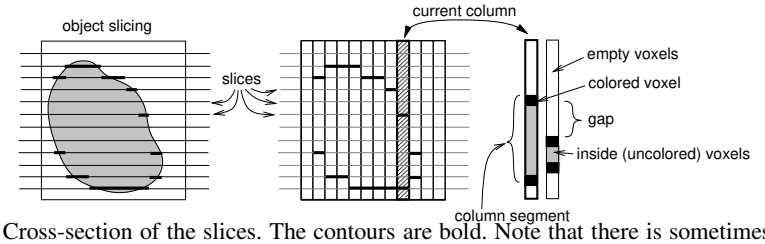


Fig. 6. Cross-section of the slices. The contours are bold. Note that there is sometimes a gap of several slices. This means that the intermediate slices have unclosed contours at this location.

The contours are generally not closed because the drawing of polygons viewed at a grazing angle (typically on the shape silhouette) generates consecutive pixels that can fall in very distant slices, i.e. their step in z is greater than the z interval between slices. This results in the contour not being drawn on the intermediate slices (see figure 6). To cope with this, we have to close the discretized shape surface by filling the gaps: let us call a *column* the set of pixels at a given location (x, y) through the set of slices. We now consider the volume as a set of columns. The segments in a column that fall inside the shape are made of uncolored pixels marked 'inside' bounded by two colored pixels, as shown in figure 6(right). A gap in the surface occurs when some of the uncolored column pixels directly neighbor the outside, because one or more of the four neighbor columns are empty at that position. Call *exposed* (i.e. to the outside) this part of the column. The problem of closing the contours is thus equivalent to filling these exposed voxels. Several algorithms are available for closing and the shape filling [10]. We have

implemented a basic algorithm for our tests, which we present in appendix. Note that the contour completion process has also to get color values for the contour pixels added, to be obtained by interpolating the surrounding colors.

Once the inside is marked and the surface is closed (i.e. there are closed contours in every slice), we fill the contours by iteratively propagating the colors. If a pixel is marked as inside and uncolored, and at least one pixel in its neighborhood is marked as colored, then the pixel is painted. The used color is the average of the colored pixels of the neighborhood (it is drawn in a separate buffer to avoid bias due to the order of scanning).

3.3 Other kinds of shape specification

Implicit surfaces

Implicit surfaces [1] are easier, because they directly specify solid objects: a pixel is inside if the implicit function is greater than one, and the shading is obtained using the gradient of the function as a normal. Since the construction of the texture pattern does not especially need to be efficient (it is a precomputation), we simply evaluate the implicit function at each volume location along the slices, and thus no filling is required. Hypertextures [12] can be handled exactly the same way.

Height fields

Height fields are a popular way of specifying the details of a surface. We consider a 2D grey-level image with $[0, 255]$ range values. Each image (x, y) location corresponds to a column in the volume: we set the pixel of the slice that fits the z value encoded by the intensity of the image at that location. Thus, the slicing stage is trivial. The normals can be computed from the height gradient in the neighborhood and used with the Phong shading to get a color.

We have used Perlin noise [11] to produce grey-level images to be used as height fields (see figure 12 in section 6). Since this function is continuous, we prefer to directly use its gradient to get the normals, thus we provide directly to the height fields voxelizor an image of the shading in addition to the image of depth. We also incorporate in this image information such as color and darkness (proportional to the depth).

The filling is similar to the process done in section 3.2 (and detailed in appendix 7) for shapes specified by their surface. It is simpler however, since each column has only one segment, with the top given by the image and the bottom at the volume bottom. The inside corresponds to all the voxels that are below the given z value. The contours have to be closed as explained in section 3.2, with fewer special cases. The final color filling of the slices is exactly the same.

4 Dealing with grazing view angles

When coping with grazing view angles, typically on the silhouette of the underlying surface on which the texels are mapped, horizontal slices are no longer satisfactory because the gap between slices appears, as illustrated in figure 4(right). In this section, we introduce quality criteria to decide if the appearance of a texel is correct or not, and we additionally store alternate directions of slice sets to get a correct appearance for any view direction. This also provides some hints for optimizations.

4.1 Quality criteria

When the view direction has an large angle from the vertical (of a texel), one can may be able to see through the sliced shape, because the projections on screen of two consecutive slices are not superimposed (see figure 4(right)). This depends on the angle a,

on the length h between slices, and on the (horizontal) thickness of the filled contours in the slices (call e the narrowest slice), as represented in figure 7(left). This provides a first quality criterion, $h\tan(a)/e \leq 1$, that indicates when such an artifact does not occur. This criterion can be used either to choose the number of slices to use to represent correctly a detail up to a given limit view angle, or to switch to an alternate slicing direction as describe in subsection 4.2.

However, as stated in section 3, the image may look degraded even for smaller view angles, because when the view direction is not orthogonal to slices one can see some pixels that are inside the contours. At some point an inside color differs from that of the surface. This provides for another quality criterion: if the user does not tolerate that the inside can be seen "deeper" than a constant d (see figure 7), the criterion is $h\tan(a)/d \leq 1$. A maximum value for d is $e/2$: since the shading of two opposite sides often has the opposite contrast, at some point (when closer to the other side of the contour) the deepest visible inside pixel color is closer to the opposite contrast than to the color of the border whose is should appear continuous (see figure 7(right)). The smallest contour thickness being e, this gives the limit of penetration $e/2$. Once again, such a criterion helps in choosing a correct number of slices. E.g. if one wants to allow view angles up to $a = \pi/3$ and the narrowest horizontal part of the shape is $e = 3$ texture pixels wide, then the distance between slices should be less than 0.9 times the length of a texture pixel.

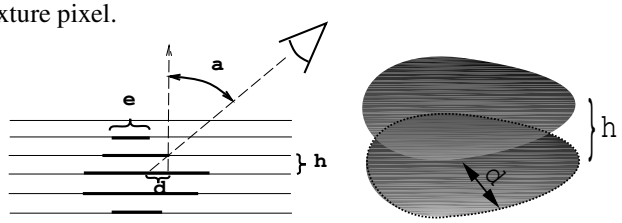


Fig. 7. *Left*: Slicing characteristics. *Right*: Limit for the second criterion: half of the inside of the narrowest slice is visible.

4.2 Alternate slice directions

To deal with grazing view angles (relative to the main slice set, called 'horizontal', i.e. parallel to the underlying surface), we also store the same volume as a set of slices in the two vertical slice directions (see figure 8).

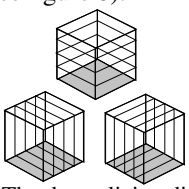


Fig. 8. The three slicing directions.

At rendering time, the dot product c_i of the three directions and the view direction indicates which of the three sets to use (a classical solution when one has to draw objects organized along a 3D grid or an octree [3]). As suggested in the previous subsection, one should choose the slices direction that has the smaller quality criterion value. The tangent of the view angle is $\sqrt{1-c_i^2}/c_i$, so that the direction to use is the i for which $(1/c_i^2 - 1) * h_i^2$ is minimum, with h_i the slice density for direction i, i.e. the size L_i of one texel in this direction divided by the number N_i of slices in this direction[5]. Thus, a

[5] However in our early implementation, we simply choose the direction for which the absolute value of c_i is maximal.

volume can be visualized correctly from any direction.[6]

4.3 Optimizations

Rendering a complex scene modeled with volumetric textures finally consists in drawing several thousands of textured polygons. There are two aspects in the rendering cost:
- the number of textured polygons that are drawn,
- the efficiency of the rendering process.

The number of polygons

There are two ways of decreasing the number of polygons to render: not drawing invisible texels (or slices), and using the minimum slice density for each texel.

The first issue is not easy to solve, since a texel lying on a back face can have some parts that are visible, so that culling is not trivial in general. A possible improvement would be to first draw (and temporarily) the bounding box of the texel, to check if at least one screen pixel was affected (e.g. using the stencil planes), and to proceed the rendering of this texel only in this case (quite like in [14] for visibility culling).

The second issue can be dealt with by deriving the minimum number of slices N_i to get a correct image from the quality criterion: $h_i \tan(a)/d \leq 1$, so $N_i \geq L_i \tan(a)/d$. This provides at the same time the direction and the number of slices to get a correct result with the lowest cost.

Another criterion can be used to decrease the number of slices with the distance: if one wants that the apparent distance between slices be less than p pixels on screen ($p \leq 1$), the criterion is $\frac{h \sin(a)}{z/f} \leq p$.

To avoid aliasing, we proceed quite similarly to MIP-mapping [13]: the number of slices in each set is a power of two, and we precompute several sets of slices. The criteria provide an optimal number of slices, which we round up to a power of two, that gives the set number to use. (None of these optimizations were used when running the tests presented in the result section.)

The efficiency of rendering

The effective rendering cost is strongly linked to the fact that the various graphics system bottlenecks can be avoided. In our case, a crucial one is the saturation of the texture cache. To minimize the potential texture cache faults, we use an alternate texel rendering method, that first draws all the occurrences of a given texel slice (in order to satisfy the back-to-front drawing requirement, the slices of front facing and back facing texels are drawn separately). Notice that this is valid only as long as there is no semi-transparent data, which would need to draw the back texels before the front texels.

5 Animating volumetric textures

Three ways of animating volumetric textures are mentioned in [7], that also correspond to three scales (illustrated on figure 9):
- deforming the underlying surface (e.g. for a flag or the skin of an animal),
- deforming the texture mapping, particularly the height vectors orientation (e.g. to simulate the wind on grass or fur),
- using several cycling volume contents along time, as for cartoons (e.g. for local oscillations).

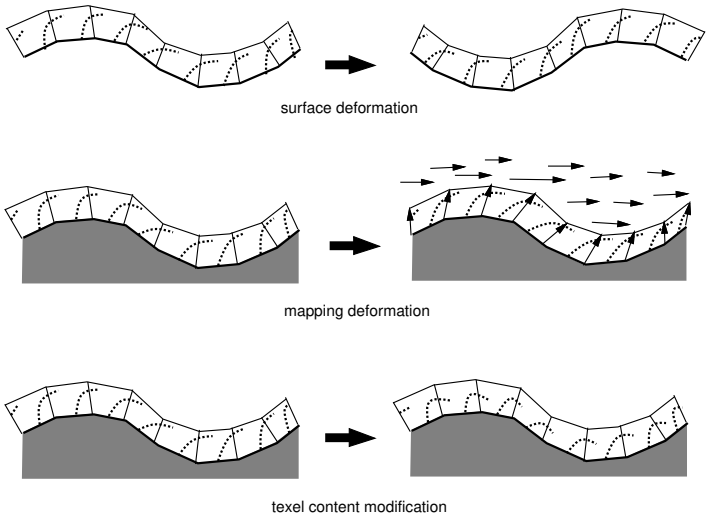


Fig. 9. The three modes of animation, that also correspond to three scales.

These methods are still usable with our interactive rendering. Surface vertices or height vectors modifications need to recompute few items at each frame, which can be done using physical models (see [7]). Time constraints are the same as for any near real-time animation of simple surfaces. Cycling a volume set has some consequences on memory if different volumes of the set are visible in the same frame. Notice that the texture memory on SGI O_2 is the same as the main memory, so that this is not really a limitation on the machine we use for our tests. However this can be a problem on other platforms, thus the drawing of the instances of a given pattern has to be grouped together, so that the texture cache changes only once per kind of pattern.

6 Results

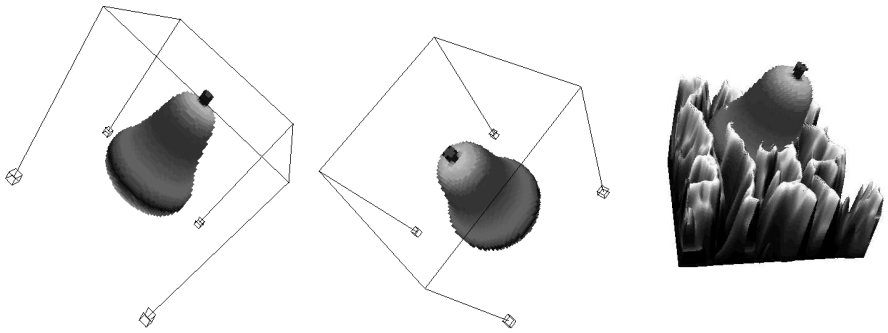


Fig. 10. A single pear texel at resolution 64^3. *Right*: Surperimposition of two texels.

The first example is an Inventor database of a pear, having about 1000 faces. The figure 10 shows a single texel at resolution $64 \times 64 \times 64$ from various viewpoints (using

[6] Notice that SGI has 3D texture facilities that also consist of a color volume, which can be indexed more easily (a single set of slices is sufficient). We have chosen not to use this feature because it is not available on most graphics hardware (in contrast to textures and Z-buffer), and because we want to control the texture memory usage to avoid swapping.

different slices directions). On the right, the figure illustrates that the superimposition of texels works correctly. In figure 11(left) we present the mapping of 96 pears on a sphere mesh having 192 triangles. At video size, this scene is refreshed at 2 frames per second on an SGI O_2. Since one texel representing the pear is rendered with 64×2 triangles, while the geometric model contains 1000 triangles, the rendering gain is about 7.5 times with equal visual complexity. Note that the pear is a simple model; the gain would be more when using a more complicated model. Oppositely, it is clear that our method is not interesting if the complexity of the pattern is less than 64 faces.

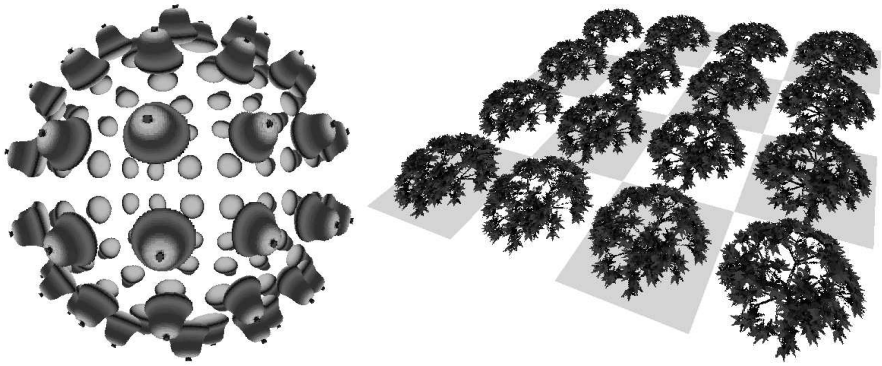


Fig. 11. Left: Mapping of 96 pear texels on a sphere. **Right:** Mapping of 16 bushes.

The second example is based on an AMAP [2] generated bush of 3,500 triangles. Since the data consists in sparse triangles, no filling is done. The texels have a $256 \times 256 \times 64$ resolution. The rendering of 16 instances shown in figure 11(right) is done at 6 frames per second. Note that because the number of instances is tuned at the mapping level, the cost would be the same even with many more bush instances.

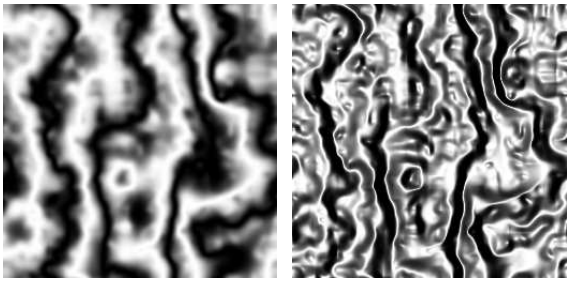


Fig. 12. Cyclical Perlin noise used to generate the height field, and the illumination computed from its gradient.

The third example uses an height field created with a cyclical Perlin noise. The noise and the illumination computed from its gradient are figured in 12. A single texel at resolution $256 \times 256 \times 64$ (i.e. with 64 slices) is shown on figure 13, with various deformations obtained (in real-time) by modifying the height vectors. Such an height field should be geometrically represented with $256 \times 256 \times 2 = 131,072$ triangles, while using this texel it is rendered with 64×2 triangles, with equivalent visual complexity. Here, the gain in polygon drawing is about 1000 times. Note that some artifacts occur on the top left of the deformed texel, where the quality criterion is not satisfied (by evaluating the criteria at the facet center, one assumes the deformation is small).

Image 15 (see color section) represents the mapping of 96 texels on a sphere mesh

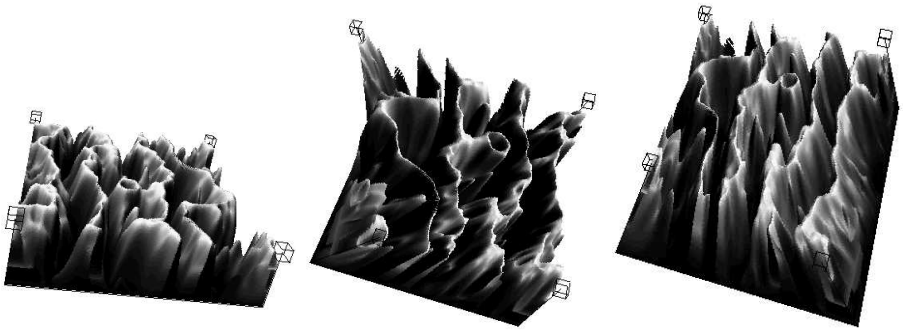


Fig. 13. *Left*: A texel at resolution $256 \times 256 \times 64$ created from the height field. *Middle and right*: Deformation of the texel by modifying the height vectors.

of 192 triangles. The frame rate is about 1.3 frames per second on an SGI O_2. The rendering of this scene could be optimized a lot, as suggested in section 4.3: no back-face culling is performed. Moreover, half of the texels appear on the sphere silhouette, thus 10% of the image represents 50% of the drawing, and 66% of the cost (because vertical slicing that is more dense due to the texture resolution is used near the silhouette). This is a waste, because on the silhouette keeping a lot of slices is useless considering the criteria seen in section 4.1. We thus expect to multiply the frame rate by about 5 by doing these optimizations. The figure 16 (see color section) illustrates the mapping of 100 texels on a jittered plane at 2.5 frames per second. This last scene has a visual complexity of 13 million of triangles.

7 Conclusion

We have presented a way to considerably increase the visual complexity of scenes displayed in the scope of interactive rendering, by adapting the ray-traced volumetric texture method [8, 9] to the graphics hardware features typically available on today's 3D graphics cards. Each texel mapped on a surface is rendered by drawing a set of extruded faces covered with transparent textures. The extrusion is controlled by height vectors located at the vertices (which can be animated). We propose several ways to build the texture content from various 3D descriptions (meshes, implicit surfaces, height fields).

Compared to the ray-tracing version, the rendering quality is of course lower (shadows and illumination are fixed). But compared to the low complexity of the scenes usually displayable at an interactive rate, our method brings a large improvement as shown by our results: the apparent complexity can be of 13 million polygons. The realism induced by the amount of visible details was previously totally unavailable for virtual reality applications. Among possible applications, we aim at introducing these apparent details in a surgery simulator we are working on. The main organ surfaces are reconstructed from scanner data, and only these surfaces are taken into account in the physical simulation of deformations. The 3D details added by the volumetric texture simply enrich the image by "dressing" these surfaces.

As future work, we want first to improve the frame rate by implementing the optimizations mentioned in section 4.3, and optimizing the OpenGL code, in order to get closer to real-time. We are currently working on the algorithm which uses an adaptive number of slices. Another issue is the development of a less naive filling algorithm to deal with more complicated patterns. We are also investigating ways of generating local illumination on the fly, possibly in the spirit of bump-mapping textures using high-end graphics capabilities.

References

1. J. Bloomenthal, C. Bajaj, J. Blinn, M.P. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, 1997.
2. Phillippe de Reffye, Claude Edelin, Jean Françon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and development. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22(4), pages 151–158, August 1988.
3. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practices (2nd Edition)*. Addison Wesley, 1990.
4. James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23(3), pages 271–280, July 1989.
5. Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear–warp factorization of the viewing transformation. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 451–458, July 1994.
6. Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993.
7. Fabrice Neyret. Animated texels. In *Eurographics Workshop on Animation and Simulation'95*, pages 97–103, September 1995.
8. Fabrice Neyret. Synthesizing verdant landscapes using volumetric textures. In *Eurographics Workshop on Rendering'96*, pages 215–224, June 1996.
9. Fabrice Neyret. Modeling animating and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), January–March 1998. ISSN 1077-2626.
10. Theo Pavlidis. *Algorithms for Graphics and Image Processing*, pages 167–193. Springer-Verlag, 1982.
11. Ken Perlin. An image synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 287–296, July 1985.
12. Ken Perlin and Eric M. Hoffert. Hypertexture. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23(3), pages 253–262, July 1989.
13. Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17(3), pages 1–11, July 1983.
14. Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Computer Graphics (SIGGRAPH' 97 Proceedings)*, pages 77–88, August 1997.

Appendix: Filling the shapes

Marking the inside of the shape

This stage prepares the color propagation stage (and can also help the contour closing stage), by indicating where to propagate. It is thus a regular filling problem. We simply consider the parity of surface crossing between the current location and the top: for each (x, y) horizontal location, we traverse the 'volume' along z (i.e. the successive slices) from top to bottom, assuming that the top is outside the shape, and we flip a flag each time an opacity transition is found at a voxel (i.e. a texture pixel of a slice). Thus the inside area of each slice is marked. This method was easy to implement for our tests, but is known to fail for complicated shapes. A better filling method like non-recursive connectivity filling [10] should better be used in general.

Closing the contours

We have seen in 3.2 that this is equivalent to filling the exposed part of a column. We interpolate the color in the intervals defined by the top of the current column segment and the top of the neighbor columns segments that start below it (and we do the same for the bottom). We compute this interpolation for each direction in which the column is exposed to the outside (i.e. up to four), and we store the mean of these, thus coloring the missed contour pixels (see figure 14(left)). If the surface is vertical, the column has no neighbor in one direction. Then we we interpolate the color from the top to the bottom of the segment (see figure 14(right)).

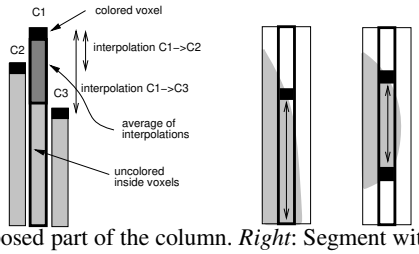


Fig. 14. Filling of the exposed part of the column. *Right*: Segment with one or two colored ends.

Fig.15. Mapping of 96 texels on a sphere.

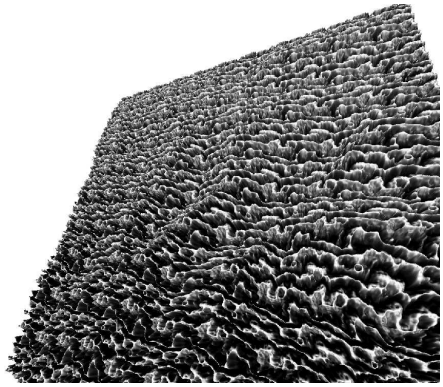


Fig.16. Mapping of 100 texels on a jittered plane made of 200 triangles.

Multiscale Shaders for the Efficient Realistic Rendering of Pine-Trees

Alexandre Meyer Fabrice Neyret

iMAGIS[†]-GRAVIR / IMAG-INRIA

Email:{Alexandre.Meyer|Fabrice.Neyret}@imag.fr

Abstract

The frame of our work is the efficient realistic rendering of scenes containing a huge amount of data for which an a priori knowledge is available. In this paper, we present a new model able to render forests of pine-trees efficiently in ray-tracing and free of aliasing. This model is based on three scales of shaders representing the geometry (i.e. needles) that is smaller than a pixel size. These shaders are computed by analytically integrating the illumination reflected by this geometry using the a priori knowledge. They include the effects of local illumination, shadows and opacity within the concerned volume of data.

Key words: Shaders, levels of details, natural scenes, ray-tracing

1 Introduction

Natural scenes such as landscapes and forests are extremely complex in term of the number of geometric primitives that lies in the field of view. Trees belongs to this category of objects that have no defined surfaces, which makes most of the geometry inside the canope potentially visible and potentially enlightened. Ray-tracing such a scene is thus very costly and very subject to aliasing. On the other hand, geometric details like needles or leaves are so small that they usually cannot be seen except for the nearest trees. Boughs of leaves themselves merge with distance. It is thus tempting to replace the indistinguishable data by a fuzzy primitive that would reproduce the same photometric behavior that the group of geometry it represents. In this paper, we propose such primitives at several scales for the particular case of the pine-tree or fir-tree. This approach can certainly be extended to other kind of trees, or to other objects for which an a priori knowledge on the shape distribution exists.

[*] *iMAGIS* is a joint research project of CNRS/INRIA/UJF/INPG. iMAGIS, INRIA Rhône-Alpes - ZIRST, 655 avenue de l'Europe, 38330 Montbonnot Saint Martin, France.

2 Previous Work

Which aspects matter in the photometric behavior of a group of shapes ? The cumulated local illumination, the cumulated shadows, and the cumulated opacity. An a priori knowledge on the matter distribution will help to compute them. Conversely, the exact shape and location of single parts are unimportant as soon as they introduce no correlation in the visibility of parts that is not already captured in the a priori knowledge. We survey now the existing models which purpose is to represent the effects of the small scales and the rendering models of trees.

Surface shaders

Some primitives have been proposed early to figure small surface details without rendering explicitly their geometry: Blinn has introduced textures of Phong parameters [3] and bump-mapping [4] in this purpose.

Kajiya has introduced the idea of hierarchy of models [11]. In this paper, he suggests to switch from geometry to mapping of Phong parameters, then to reflectance model[1] according to the distance. Transitions from geometry to bump and from bump to reflectance have been proposed in [1, 5, 7].

Several reflectance models based on the surface micro-geometry have been developed [11, 24, 8, 17, 6, 10, 9]. Most of these models consist in proposing a representation of the matter distribution, then to integrate the local illumination while addressing the visibility of the details for the viewer and for the light (i.e. self-shadows).

Volume shaders

All the models above are designed for surface details. In the scope of 3D matter distributions, Blinn has early proposed a reflection model for volumes of dust [2] represented by micro-spheres. Stam has developed in [21] a stochastic model which allows the analytical integration of the stochastic distribution of matter to represent details in clouds. Kajiya introduced the volumetric textures [12]

[1] reflectance models are also named *shaders*.

in the scope of fur rendering. A shader (i.e. a local illumination model) is derived to integrate the light reflected on hairs represented by cylinders. This cylinder shader has been improved in [8] and Neyret has extended the volumetric textures representation in [14, 15] by introducing a shader able to integrate at one scale the shaders representing a thinner scale.

Contrary to the models of surface details, most of the 3D models presented above fail to address analytically the visibility of the details from the viewer or from the light. For instance the representation of the 3D microgeometry by a normal distribution in [14] cannot capture the visibility (otherwise the normal distribution should depend on the point of view), whereas the stochastic model of [22] can.

Dedicated tree rendering models

On the other hand, several models dedicated to an efficient representation and rendering of trees and forests have been proposed, using ray-tracing or real-time techniques [16, 23].

Reeves introduced the particles systems [19, 20]. This representation is dedicated to objects made of a huge amount of small long primitives that are drawn as simple strokes, well suitable for modeling trees. In his paper, the shadows are faked using a priori simple laws such as proportionality with depth inside a tree.

Max proposed in [13] a hierarchical representation of trees based on color-depth textures following the natural hierarchy of trees.

Early conclusions

To conclude at that point, we can tell that:

- shaders based on a normal distribution function difficultly account for the shadowing inside the small scale.
- shaders consisting in a sampled BRDF are more accurate, but cannot easily be parameterized.
- shaders consisting in analytical BRDF can be both visibility-compliant and parameterized but are not easy to derive.

Our key idea is that such analytical BRDF can be derived when strong a priori knowledge on the matter distribution is available. We think that the matter in trees is structured enough to offer such a possibility. In this paper we address needle-based trees such as pine-tree or fir-tree, as the a priori knowledge on needles distribution is strong.

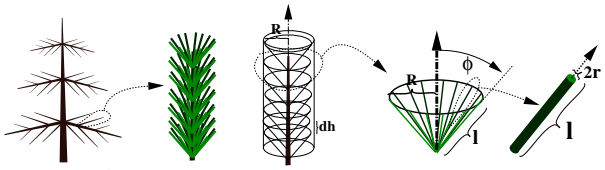


Figure 1: Our hierarchical description of a tree.

3 Contributions

3.1 Our model of pine-tree (see Figure 1)

- A tree is a set of branches and needles that we describe using an L-system [18].
- Branches are classical geometry (i.e. cylinders).
- Needles are cylinders, whose angle ϕ with the branch, length l, radius r, density (i.e. distribution) ρ change slowly along a branch so that they can be considered locally constant.
- The needles layer around the branch (i.e. the bough) is thus a cylinder of radius $R = l\sin(\phi)$.
- We assume that needles are spreaded on cones, with N needles per cone. The distance between cones along a branch is dh. As the gap between two needles end is $\frac{2\pi R}{N}$ and the gap between two cones is dh, it is reasonable to chose $dh = \frac{2.\pi.R}{N} = \sqrt{\rho}$. Whether it is the case or not, we have the relation $dh\frac{2\pi R}{N} = \rho$.

3.2 Multiscale rendering

Depending on the distance, the smallest primitive we consider is either the needle (level 1), the cone (level 2), or the bough (level 3). We render the scene using a simple cone-tracing: the conic ray is used to estimate the apparent size of primitives and to compute their coverage $alpha$ to the pixel. We also use this cones for the shadow rays, assuming point light sources.

The main issue is to compute the global reflectance and opacity of a considered primitive, including the internal shadows. Since we use only conic rays, the rendering is processed with no oversampling at all.

Thus the main contributions of this paper are the multiscale representation that we detail in the next section, the three shaders we derive (detailed in sections 4, 5 and 6), and the method we use to solve the illumination integrals, in particular the geometric interpretation of the visibility and shadows in the level 3 model.

3.3 What we need to compute

In this section we estimate the requirement for the analytical computation of the three shaders. The results and the details of these successive integrations are the object of the three next sections. The $\vec{L}$ and $\vec{V}$ vectors are considered constant because light source and the viewer are far.

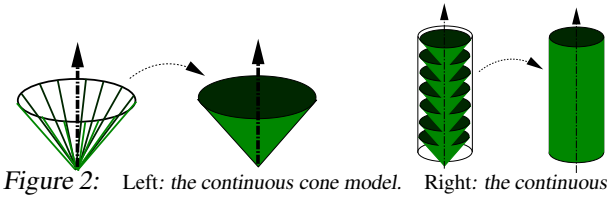


Figure 2: Left: *the continuous cone model.* Right: *the continuous bough model.*

Level 1 (needles)

To shade a needle, we need the amount of diffuse and specular light I_d and I_s reflected by a cylinder [12]. In [17] the integral is correctly expressed in pixel space. We use this form, with different bounds and a cheaper approximation for the specular integral.

We never compute explicitly the intersection of the needles with the cone-ray. We compute instead the intersection of a cone of needles, and we consider the needles that are on the visible part of the cone. Then we sum their illumination.

Level 2 (cones)

We consider that shading a cone of needles is equivalent to shading a continuous semi-opaque cone whose each point reflects the light as a local needle would (see Figure 2 left). The opacity A is the amount of the cone surface covered by needles, so is defined by $A = \frac{2Nr}{\pi R}$. The illumination is A times the integral in pixel space of the cylinder illumination on the visible part of the cone. The front and rear part are considered separately, and only a portion of these parts may be visible in a pixel. This integration is not trivial and requires several approximations.

Level 3 (boughs)

We consider that a bough to be shaded is equivalent to a semi-opaque anisotropic volumetric cylinder made of imbricated cones (see Figure 2 right). The illumination and opacity of front and rear parts of the cones correspond to the level 2 shader already derived (the front part of all the cones are equal, same for the rear parts). The volume model is both continuous and anisotropic: the opacity has to reproduce the same effect as the number of cones traversed by a ray while rendering at level 2, which is strongly dependent of the angle of the ray. The difficult part is the analytical volumetric integration of it, taking into account the visibility and the shadows. Assuming we can use a linear approximation [2] of the opacity composition law, i.e. $(1-A)^n \approx (1-n.A)$, we transpose this integral into a geometric form.

4 Cylinder illumination

We have to integrate the diffuse and specular components into screen space (i.e. we sum the contributions to the pixel color and opacity). Either reflectance or illumination can be derived; one can trivially convert one into the other since we also compute the opacity.

[2] which is valid for $nA \ll 1$, i.e. if the bough is not too dense

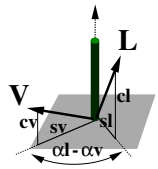


Figure 3: *A single needle.*

- The diffuse reflectance toward the viewer is

$$R_d^{cyl} = \frac{\int_{cylinder} (N.L)\,\mathbb{1}_{(N.L>0)}\,(N.V)\,\mathbb{1}_{(N.V>0)}.dS}{\int_{pixel} (N.V)\,\mathbb{1}_{(N.V>0)}.dS}$$

Let c_v and c_l be the projections of V and L on the cylinder axis $\vec{a}$, i.e. $c_v = (\vec{a}.V)$ and $c_l = (\vec{a}.L)$ (see Figure 3).
Let V_p and L_p be the projections of V and L on the plane orthogonal to the cylinder, and s_v and s_l be their norm.

$$R_d^{cyl} = \frac{\int_{\alpha=\alpha_0}^{\alpha_1} s_l \cos(\alpha-\alpha_L) s_v \cos(\alpha-\alpha_V) d\alpha}{\int_{\alpha=\alpha_V-\frac{\pi}{2}}^{\alpha_V+\frac{\pi}{2}} s_v \cos(\alpha-\alpha_V) d\alpha}$$

with α_V and α_L the angles between a reference in the plane and respectively V_p and L_p. The bounds of visibility α_0 and α_1 are $\alpha_V - \frac{\pi}{2}$ and $\alpha_L + \frac{\pi}{2}$ if $L \times V$ has the same direction than $\vec{a}$. We introduce $\Delta\alpha = |\alpha_V - \alpha_L|$ and then

$$\boxed{R_d^{cyl} = \tfrac{s_L}{4}(\sin(\Delta\alpha) + (\pi - \Delta\alpha)\cos(\Delta\alpha))} \quad (1)$$

- The specular reflectance toward the user is

$$R_s^{cyl} = \frac{\int_{cylinder} (N.H)^n\,\mathbb{1}_{(N.H>0)}\,(N.V)\,\mathbb{1}_{(N.V>0)} dS}{\int_{pixel} (N.V)\,\mathbb{1}_{(N.V>0)} dS}$$

with the half-way vector $H = \frac{V+L}{|V+L|}$ and n the specularity exponent.
Let H_p, c_h, s_h and α_V be defined like for L and V. Then

$$R_s^{cyl} = \frac{\int_{\alpha=\alpha_0}^{\alpha_1} s_h^n \cos^n(\alpha-\alpha_H) s_v \cos(\alpha-\alpha_V) d\alpha}{\int_{\alpha=\alpha_V-\frac{\pi}{2}}^{\alpha_V+\frac{\pi}{2}} s_v \cos(\alpha-\alpha_V) d\alpha}$$

It is well known that $\cos^n(x)$ is very similar to $e^{-\frac{n}{2}x^2}$ for n large (which is the case). Moreover the density of this function is concentrated on $x=0$ (the standard deviation is $1/\sqrt{n}$, and n is generally greater than 100), so that $\cos^n(x-x_0) f(x) \approx \cos^n(x-x_0) f(x_0)$
Thus, we have
$R_s^{cyl} \approx (s_h^n s_v \cos(\alpha_H - \alpha_V) \int_{\alpha=\alpha_0}^{\alpha_1} e^{-\frac{n}{2}(\alpha-\alpha_H)^2} d\alpha)/2s_v$
Since $\int_{-\infty}^{\infty} e^{-\frac{1}{2}(\frac{x}{\sigma})^2} = \sqrt{2\pi}\sigma$, the integral above equals

$\sqrt{\frac{2\pi}{n}}$ if $\alpha_H \in [\alpha_0, \alpha_1]$ which is always the case. Thus

$$\boxed{R_s^{cyl} \approx \frac{1}{2} s_h^n \cos(\alpha_H - \alpha_V) \sqrt{\frac{2\pi}{n}}} \quad (2)$$

• The opacity is the proportion of the needle apparent rectangle that falls in the pixel. If the needle is totally covered by the pixel, then

$$\boxed{alpha^{cyl} = \frac{2r.s_v.l}{S_{pix}}} \quad (3)$$

where S_{pix} represents the surface of the ray-cone section at the primitive's distance. Thus the diffuse and specular illumination are $I_d = alphaR_d$ and $I_s = alphaR_s$.

5 Cone illumination

As discussed in section 3.3, we consider that the cone is a continuous semi-opaque surface of opacity A, whose each point of the surface reflects the light as a cylinder. Thus, we need to integrate the cylinder illumination into a cone of aperture ϕ for all the valid needle axis positions $\vec{a}_\theta$. In the polar coordinate system associated to the cone, we denote $L = (\theta_L, \phi_L)$, such that ϕ_L is the angle between L and the cone axis. Similarly we denote $V = (\theta_V, \phi_V)$.

• The diffuse illumination is given by:

$$I_d^{cone} = \frac{lA}{4} \int_{\theta=\theta_V-\frac{\pi}{2}}^{\theta_V+\frac{\pi}{2}} s_l s_v \left(\sin(\Delta\alpha) + (\pi - \Delta\alpha)\cos(\Delta\alpha)\right)$$

where ls_v is the apparent length of a needle.

We cannot integrate analytically this formula. As such, we approximate $s_l s_v(\sin(\Delta\alpha) + (\pi - \Delta\alpha)\cos(\Delta\alpha))$ by using the function

$$F = s_l s_v \left(1/2 + \cos(\Delta\alpha)/2\right)\left(2 + (\pi - 2)\cos(\Delta\alpha)\right)$$

which has the same values and derivatives in 0, $\frac{\pi}{2}$ and π and which maximum error is less than 1%.
Since $\cos(\Delta\alpha) = \frac{(L_p.V_p)}{(|L_p|.|V_p|)} = \frac{(L.V) - c_l c_v}{s_l s_v}$ then
$\int F = (L.V + s_l s_v - c_l c_v).(2 + (\pi - 2)(L.V - c_l c_v)/s_l s_v)$

Figure 4: Left: *An example of F curve, for* $L = (0, 1.2)$, $V = (1, 1.5)$ *and* $\phi = .5$. *It is very smooth, despite its factors are quite more chaotic.* Right: *the FFT of this curve. Notes that the energy is clearly concentrated on the frequencies 0, 1 and 2, thus the motivation to fit F with a linear combination of 1,* $\cos(\theta - \theta_A)$, $\cos(2(\theta - \theta_B))$. *NB: the values at the extreme right are the mirroring due to the FFT.*

When tracing this function with Maple for many values of the parameters L, V and ϕ, it appears that the curve is very smooth (Figure 4 left), and looks like a linear combination of 1, $\cos(\theta - \theta_A)$ and $\cos(2(\theta - \theta_B))$. The FFT evaluation on discretized curves shows that there is practically no energy out of the frequencies 0, 1 and 2 (Figure 4 right). As such, we try to fit such a curve to F from the location and value of its extrema. The first factor capture most of the variations of F and is more easy to analyze, so to fit the curve we approximate F by $(L.V) + s_l s_v - c_l c_v$ which seems to have its extrema at the same θ value than F.
The term $c_l c_v - s_l s_v$ equals $\cos(\widehat{AL} + \widehat{AV})$ with $\widehat{AL}$ the angle between the vectors $\vec{a}$ and L , and $\widehat{AV}$ the angle between the vectors $\vec{a}$ and V. These angles vary smoothly between a minimum and a maximum while $\vec{a}$ rotates along the cone, so we model the variation of $\widehat{AL}$ by the form $A_L + B_L\cos(\theta - \theta_L)$ with $A_L = \max(\phi_L, \phi)$, $B_L = \min(\phi_L, \phi)$. We do the same for $\widehat{AV}$.
If we develop $\widehat{AL} + \widehat{AV}$ with this approximation we obtain the expression $A_\Sigma + B_\Sigma\cos(\theta - \theta_\Sigma)$ with
$A_\Sigma = A_L + A_V, \quad B_\Sigma^2 = B_L^2 + B_V^2 + 2B_L B_V \cos(\theta_L - \theta_V)$,
$\cos(\theta_\Sigma) = (B_L\cos(\theta_L) + B_V\cos(\theta_V))/B_\Sigma$,
$\sin(\theta_\Sigma) = (B_L\sin(\theta_L) + B_V\sin(\theta_V))/B_\Sigma$

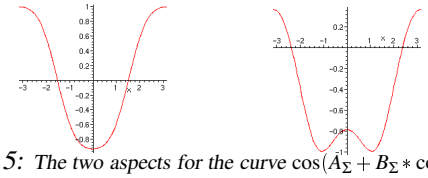


Figure 5: *The two aspects for the curve* $\cos(A_\Sigma + B_\Sigma * \cos(\theta - \theta_\Sigma))$, *depending whether* $A_\Sigma + B_\Sigma * \cos(\theta - \theta_\Sigma)$ *crosses* π (right) *or not* (left).

We can now search for the extrema of $F \approx (L.V) - \cos(\widehat{AL} + \widehat{AV})$. They correspond either to the extrema of $\widehat{AL} + \widehat{AV}$ or to the location for which $\widehat{AL} + \widehat{AV}$ crosses π. If $\widehat{AL} + \widehat{AV}$ does not cross π, F looks like a cosine function. If it does, F has a hat shape and looks like the combination of a cosine and a cosine at double frequency (see Figure 5). The similarity is high if ϕ_L and ϕ_V are not very close to ϕ. Thus, we can now obtain explicitly the extrema of the curve.
As we are precisely trying to fit F to the form $(L.V) - (\lambda_0 + \lambda_1\cos(\theta - \theta_m) + \lambda_2\cos(2(\theta - \theta_m)))$ we just have to set the parameters from these extrema: let $M = \cos(A_\Sigma - B_\Sigma)$ and $m = \cos(A_\Sigma + B_\Sigma)$.
Then $\theta_m = \theta_\Sigma$, $\lambda_1 = \frac{(m-M)}{2}$, $\lambda_0 = \frac{(m+M)}{2} - \lambda_2$, with $\lambda_2 = 0$ if no crossing of π occurs (both $A_\Sigma + B_\Sigma$ and $A_\Sigma - B_\Sigma$ are in $[0, \pi]$),
$\lambda_2 = \frac{\lambda_1 B_\Sigma}{4(2\pi - A_\Sigma)}$ in case of crossing of π ($A_\Sigma + B_\Sigma > \pi > A_\Sigma - B_\Sigma$), Now we can easily obtain the integral of F :

$$\boxed{I_d^{cone} = \frac{lA}{4}\left(\pi(LV - \lambda_0) - 2\lambda_1\cos(\theta_V - \theta_\Sigma)\right)} \quad (4)$$

where $\cos(\theta_V - \theta_\Sigma) = \frac{B_L\cos(\Delta\theta) + B_V}{\sqrt{B_L^2 + B_V^2 + 2B_L B_V\cos(\Delta\theta)}}$ and $\Delta\theta = \theta_L - \theta_V$.

• The specular illumination is given by

$$I_s^{cone} = \frac{lA}{2}\sqrt{\frac{2\pi}{n}} \int_{\theta=\theta_V-\frac{\pi}{2}}^{\theta_V+\frac{\pi}{2}} s_l s_h^n \cos(\alpha_H - \alpha_V)$$

with ls_v the apparent length of a needle. Once again, s_h^n is a function which density is concentrated on the location where $s_h = 1$, which occurs when $c_h = 0$, i.e. when H is orthogonal to the needle direction $\vec{a}$. Such a location $\theta_H^\perp$ only exists if $\phi_H \in [\frac{\pi}{2} - \phi, \frac{\pi}{2} + \phi]$, otherwise $I_s^{cone} = 0$.

If $\theta_H^{\perp}$ exists, we have again that $s_h^n f(\theta) \approx s_h^n f(\theta_H^{\perp})$.
Since $s_l s_h \cos(\alpha_H - \alpha_V) = (V.H) - c_h c_v$, we finally have

$$\boxed{I_s^{cone} \approx \frac{l.A}{2}\frac{2\pi}{n}(V.H)\varepsilon} \quad (5)$$

where $\varepsilon = 1$ if $\phi_H \in [\frac{\pi}{2} - \phi, \frac{\pi}{2} + \phi]$ otherwise $\varepsilon = 0$.
Note that if both locations where H is orthogonal to $\vec{a}$ occurs on the same face (front or rear), we have that $\varepsilon = 2$.

- The opacity is given by $alpha^{cone} = A\int_{\theta=\theta_V-\frac{\pi}{2}}^{\theta_V+\frac{\pi}{2}} l s_v$

Since $s_v = \sin(\widehat{AV})$, we approximate $\widehat{AV}$ by $A_V + B_V\cos(\theta - \theta_V)$ in the same way that for the diffuse component. That is,

$$\boxed{alpha^{cone} = l.A(\pi\cos(\phi)\cos(\phi_V) - 2\sin(\phi)\sin(\phi_V))} \quad (6)$$

6 Bough illumination

As stated in section 3.3, we consider that the bough is a volume having a cylindrical shape and an anisotropic opacity (as illustrated in Figure 6). We have to proceed to the analytical volume rendering of this cylinder.

Since the opacity A is not constant along the ray and the shadow ray, we have:

$$I = \frac{1}{S_{pix}}\int_{(x,y)\in pixel}\int_{z=near}^{far} A I^{cyl} e^{-\int_0^{l_z}\sigma} e^{-\int_0^{l_{shad}}\sigma} \quad (7)$$

with $e^{-\sigma} = T = (1-A)$ the anisotropic transparency, l_z the length of the ray within the volume and l_{shad} the length of the shadow ray within the volume.
We need now to explicit the opacity and to do some approximations to make the integral tractable.

6.1 Traversal of a 2D bough

Given an infinite 2D vertical field of parallel needles having a direction ϕ relatively to the top (see Figure 7 left). Let R be the field width, and dh the vertical distance between the needles. A ray in the direction ϕ_r relative to the top crosses the field.
The length of the ray within the field is $R/sin(\phi_r)$
The step between the intersections is $\delta = dh\frac{\sin(\phi)}{\sin|\phi_r-\phi|}$
The average number of intersections is $\frac{R}{dh}\frac{\sin|\phi_r-\phi|}{\sin(\phi)\sin(\phi_r)}$
We denote $k(\phi_r,\phi)$ the quantity $\frac{\sin|\phi_r-\phi|}{\sin(\phi)\sin(\phi_r)} = |\frac{1}{\tan(\phi)} - \frac{1}{\tan(\phi_r)}|$
The opacity of the field along this ray is $1 - T^{\frac{R}{dh}k(\phi_r,\phi)}$
Let denote for short $k_r = k(\phi_r,\phi)$ and $\overline{k}_r = k(\phi_r,\pi-\phi)$
$\overline{k}_r$ corresponds to the traversal of a field which is symmetrical to the first relatively to the vertical.

A 2D bough is composed of two adjacent such fields, the right one with needles of orientation ϕ, and the left one with needles of orientation $\pi - \phi$ (as illustrated on Figure 7 left).
The total number of intersections along a ray is
$\frac{R}{dh}(k_r + \overline{k}_r) = \frac{R}{dh}\frac{\sin|\phi_r-\phi|+\sin|\phi_r+\phi|}{\sin(\phi)\sin(\phi_r)} = \frac{R}{dh}\frac{2}{\tan(\min(\phi,\phi_r))}$
This means that as long as the ray remains outside the cone aperture (i.e. $\phi_r \in [\phi, \pi-\phi]$) the total opacity along the ray is constant, despite it is balanced differently between the front and the rear part. This is true either for a ray or a shadow ray: similarly for the light, in such condition the shadow casted by the bough is constant, while the light enters more easily in one side than in the other. If the ray is inside the cone aperture (above or below), the opacity increases up to 100% for $\phi_r = 0$ or π.

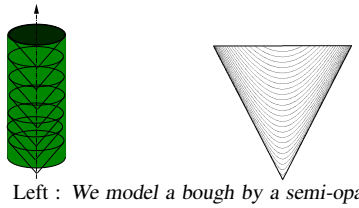


Figure 6: Left : *We model a bough by a semi-opaque volumetric cylinder, which opacity is anisotropic in order to reproduce the variation of the number or intersection between a ray and the sub cones.* Right : *Intersection of the plane P_x with one cone. We approximate the hyberbols by their asymptotes.*

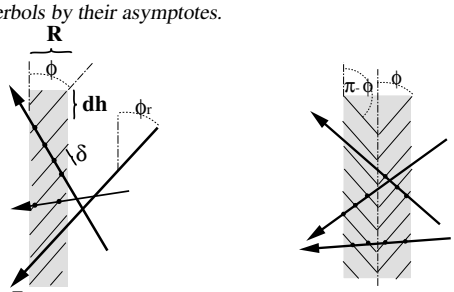


Figure 7: Left: *2D field of parallel 'needles'.* Right: *2D bough. Note the variation of the opacity with the ray direction (mostly on left).*

6.2 Extension to a 3D bough

Let us now come back to our regular bough. In 3D, if a ray crosses the axis of the bough, the situation is equivalent to the 2D situation above. But generally the ray does not cross the axis. Let consider the plane parallel to the cone axis and that contains the ray. Let x be its distance to the axis, thus we name the plane P_x. The intersection of the volume of the bough made of cones with the plane gives a set of hyperboles. We approximate these hyperboles by their two asymptotes (we can see on Figure 6 right that it is reasonable). In that way, the plane contains 'needles' having the same orientation ϕ and offset dh than in 3D, in a field of shrieked thickness $2R_x$ with $R_x = \sqrt{R^2 - x^2}$. So we can compute the number of intersections using the 2D formulas. To estimate the amount of light reaching a point on the ray, we consider a shadow ray starting at that point. Similarly, we introduce the plane parallel to the cone axis and that contains the shadow ray (Figure 8). The number of intersections can be obtained as for the main ray.

6.3 Traversal of a 3D bough

We can now come back to the volumetric integral 7. We choose the (x,y) pixel-surface parameterization so that

the $\vec{x}$ axis is orthogonal to the cylinder. Thus x indexes the plane P_x (i.e. x is coherent with the previous section). In consequence we no longer need to integrate along the $\vec{y}$ axis, since the cylinder is homogeneous in this direction. Note that the albedo A in the equation should be corrected to A/δ, since no energy is gathered in the gap between two cones. Similarly on a differential length dl, the opacity is $e^{-\sigma dl} = T^{dl/\delta}$. We proceed to a variable change from (x,z) to (x,z') in the plane orthogonal to the cylinder. This means that we index a point on the ray by its projection on the orthogonal plane. The Jacobian of the transform is $\frac{1}{\sin(\phi)}$. The opacity associated to a differential length dl' on the plane is $T^{\frac{dl'}{\sin(\phi)\delta}} = T^{dl'\frac{k()}{dh}}$

6.4 Splitting the integral into regions

We know from the 2D case that the opacity along the ray is constant on the front half and on the rear half of the traversal (These two halves correspond to the two orientations of the needles in the plane P_x).

The disk has been split into two regions F_V and R_V, the front and the rear relatively to V. On each region $k()$ is constant. In section 5 we have also split the cones into a front face and a rear face, to evaluate the illumination. Let assume that I^{cyl} is constant in each of the two regions of the volume and let approximate it by the mean value I^{cyl}_{front} and I^{cyl}_{rear}. The integral becomes:

$$I = \frac{A}{2Rdh}\int_{x=-R}^{R}\left(k_v I^{cyl}_{front}\int_{z=-R_x}^{0} T^{\frac{k_v}{dh}(R_x+z)} T^{\int_0^{l_{shad}}\frac{k()}{dh}} + \overline{k}_v I^{cyl}_{rear}\int_{z=0}^{R_x} T^{\left|\frac{k_v}{dh}R_x+\frac{\overline{k}_v}{dh}z\right|} T^{\int_0^{l_{shad}}\frac{k()}{dh}}\right)$$

In order to get rid of the remaining integral in the exponent, we are now going to split again the disk to separate the front and the rear areas F_L and R_L relatively to L. However the shadow ray length that will appear depends on z on a complicated way, which makes the exponential tricky to integrate analytically.

In order to make the integral tractable, we use the linear approximation of the opacity composition law,

i.e. $(1-A)^n \approx (1-nA)$ which is valid if $nA \ll 1$, i.e. if the bough is not too dense.

Then $(1-A)^{n_1}(1-A)^{n_2} \approx 1-n_1A-n_2A$, which ensures the separation of the factors. Thus the integral is defined as $I = I_{F_V} + I_{R_V} = \frac{A}{2Rdh}\left(I^{cyl}_{front}k_v I_{F'_V} + I^{cyl}_{rear}\overline{k}_v I_{R'_V}\right)$ with

$$I_{F'_V} = \int_{F_V} 1 - A\int_{F_V}\frac{k_v}{dh}(R_x+z) - A\int_{F_V\times R_L} z_{shad}\frac{\overline{k}_l}{dh} - A\int_{F_V\times F_L} z_{shad}\frac{k_l}{dh}$$

$$I_{R'_V} = \int_{R_V} 1 - A\int_{R_V}(\frac{k_v}{dh}R_x + \frac{\overline{k}_v}{dh}z) - A\int_{R_V\times R_L} z_{shad}\frac{\overline{k}_l}{dh} - A\int_{R_V\times F_L} z_{shad}\frac{k_l}{dh}$$

with $F_V \times R_L$ the region in R_L covered by shadow rays which origin is in F_V, and so on for the other composed regions (see on Figure 8 the representation of these surfaces).

6.5 Geometric integration

We can arrange this as:

$$I_{F'_V} = \pi\frac{R^3}{2} - A\frac{k_v}{dh}\frac{2}{3}R^3 - A\frac{\overline{k}_l}{dh}\int_{F_V\times R_L} z_{shad} - A\frac{k_l}{dh}\int_{F_V\times F_L} z_{shad}$$

$$I_{R'_V} = \pi\frac{R^2}{2} - A\frac{k_v}{dh}\frac{4}{3}R^3 - A\frac{\overline{k}_v}{dh}\frac{2}{3}R^3 - A\frac{\overline{k}_l}{dh}\int_{R_V\times R_L} z_{shad} - A\frac{k_l}{dh}\int_{R_V\times F_L} z_{shad}$$

The four remaining integrals sum the length of the shadow rays starting in each point along the ray and included in the region in subscript, for each ray. Let consider for the moment only the integral along the ray. The shadow-ray sweeps an area while its origin follows the ray. The integral of its length value along the main ray has a strong connection with this surface: it is proportional to it, with a factor $\frac{1}{\sin(l_v)}$ where l_v is the angle between the projections L_P and V_P of L and V in the orthogonal plane. The proof is that if L_P is orthogonal to V_P, then the integral of the length is the regular surface measurement. Otherwise one can come back to this case with a change of variables, which the Jacobian is $\frac{1}{\sin(l_v)}$. So, to compute the integral along the ray, we have to measure the surface of each swept region S_1, S_2, S_3, S_4 using some geometric and trigonometric relations. Then we have to integrate the result for each ray. After some long and unpleasant derivations showing quite complicated formulas in the intermediate stages, we surprisingly found very simple and symmetric results (without any approximation):

$$\int S_1 = (1+\cos(l_v))\frac{R^3}{3}\sin(l_v)$$
$$\int S_2 = (1-\cos(l_v))\frac{R^3}{3}\sin(l_v)$$
$$\int S_3 = (1+\cos(l_v)/3)R^3\sin(l_v)$$
$$\int S_4 = (1-\cos(l_v)/3)R^3\sin(l_v)$$

The $\sin(l_v)$ factors disappear when multiplying by the Jacobian.

6.6 Resulting bough illumination

The opacity is derived trivially:

$1-\alpha_{F_V} = \frac{1}{2R}\int_{x=-R}^{R} A^{R_x\frac{k_v}{dh}} \approx 1 - \frac{AR}{dh}\frac{\pi}{4}k_v$

i.e. $\boxed{\alpha_{F_V} = ak_v\ ,\ \ \alpha_{R_V} = a\overline{k}_v}$ with $a = \frac{AR}{dh}\frac{\pi}{4}$

We introduce similarly the opacity for the light point of view: $\alpha_{F_L} = ak_l$, $\alpha_{R_L} = a\overline{k}_l$ and finally have $I = I_{F_V} + I_{R_V}$ with

$$\boxed{I_{F_V} = I^{cyl}_{front}\alpha_{F_V}\left(1 - \frac{8}{3\pi^2}\left(2\alpha_{F_v} + (1-\cos(lv))\alpha_{R_L} + (3-\cos(lv))\alpha F_L\right)\right)}$$

$$\boxed{I_{R_V} = I^{cyl}_{rear}\alpha_{R_V}\left(1 - \frac{8}{3\pi^2}\left(4\alpha_{F_v} + 2\alpha_{R_v} + (1+\cos(lv))\alpha_{R_L} + (3+\cos(lv))\alpha F_L\right)\right)}$$

We leave this formula into two separated parts, which allows to render a branch between them.

7 Results

Some resulting images are presented of Figure 9 and Figure 10. We have also compute an animation of the forest scene showing no aliasing artifact.

A major property of our model is the evolution of the cost when the number of needles vary, i.e. the complexity analysis in function of the number N of needles per cone and of the number $\frac{l}{dh}$ of cones on a branch per unit of length (these two numbers are proportional to the square

root of the density of needles)[3]. The cost of one shadow ray should evolve the same. However a classical ray-tracer launches a shadow ray for each sample, while for our model the part of the shadow ray that is outside the bough is factorized.

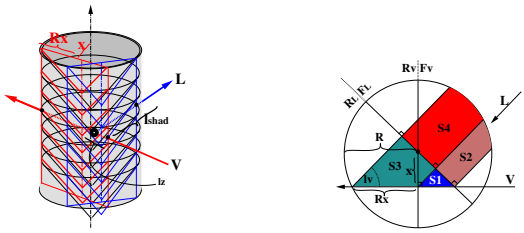


Figure 8: Left: *The volume intersected by the vertical plane containing the ray looks like a 2D bough. Similarly for the shadow ray.* Right: *The volume of the bough cylinder seen in an orthogonal section. The surface of the four regions (see left Figure)* $S_1 = F_V \times R_L$, $S_2 = F_V \times F_L$, $S_3 = R_V \times R_L$, $S_4 = R_V \times F_L$ *are proportional to the integral of the length of the shadow rays for each possible origin on the ray (only the generic case is figured here). We have to integrate these surfaces for all* x.

We have compared the efficiency to a classical ray-tracer, Rayshade. On Rayshade side, it is important to know that there is a maximum amount of ray per pixel (which is 64), so that when a tree is far (i.e. less than 100 pixels high), Rayshade does not launch enough rays. It might seems efficient, but this is at the price of quality. The fact is that for a image with a lot of high frequencies as image of trees are, the aliasing is not very visible on a single image because it is hard to distinguish noise and information. But the aliasing is obvious during an animation.

The test scene consists of 80 fir-trees that are about 127 pixels high for the closest and 64 for the farthest (Figure 11).

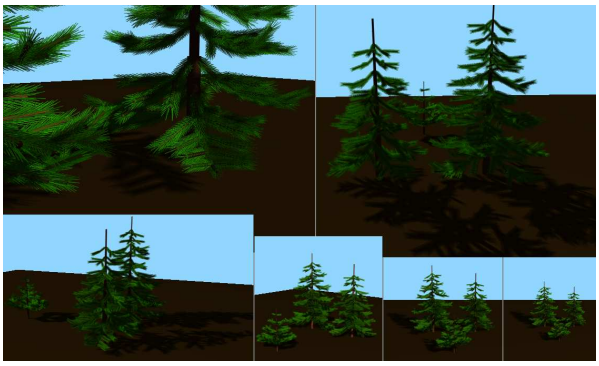


Figure 9: *Three fir-trees, from a very close to a far point of view.*

[3] if N is multiplied by 2, the number of intersections for level 1 and the number of samples per pixel a ray-tracer should launch are multiplied by 2, while level 2 and level 3 are not affected at all. The same deduction could be done if dh is divided by 2

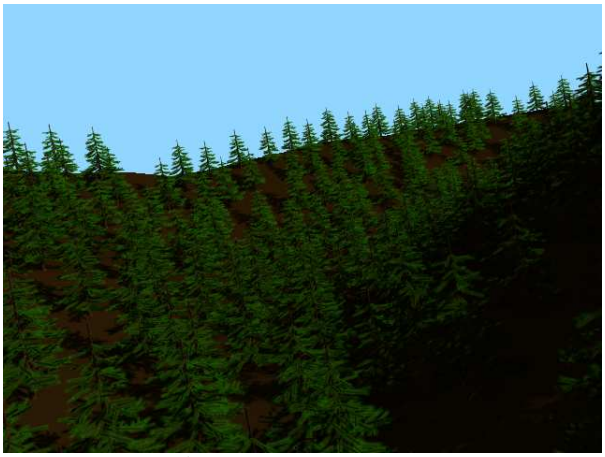


Figure 10: *Trees on a hill.*

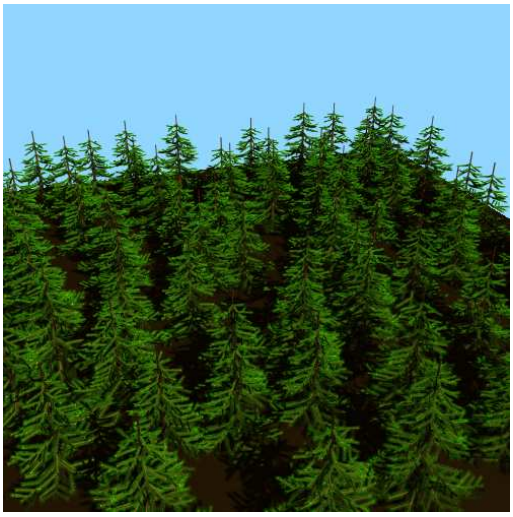


Figure 11: *The scene used for the benchmark.*

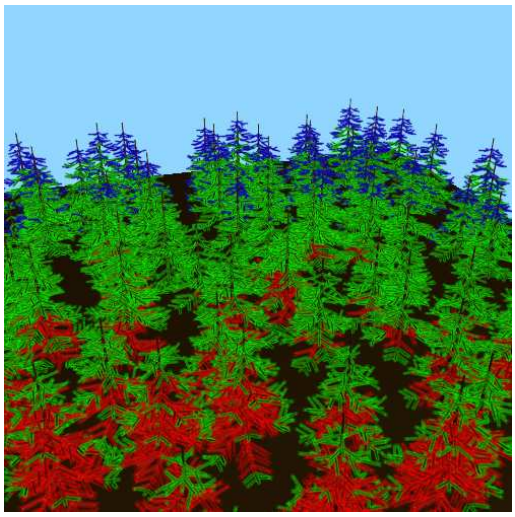


Figure 12: *The colors represent the level that is used in our method: red for level 1, green for level 2 and blue for level 3.*

One fir generally contains 300 branches and about 28700 needles, thus the scene contains about 2 million of needles. Concerning one bough, a cone is 3.94 high, has a radius of 1.6cm, an aperture of $\pi/8$, and the offset between cones is 0.9cm. There are only 12 needles per cone for this tree, whose radius is 0.05cm and length is 4.25cm. On average 4.37 cones are imbricated, so that a ray passing through the axis and orthogonal to the branch would traverse on average 8.75 layers. We run our tests on an SGI Onyx2. The rendering time is 65.3 minutes with Rayshade and 8.1 minutes with our models. Thus our method is about 8 times faster than Rayshade. For landscapes, whose farthest trees are very small, rayshade cannot avoid aliasing due to its 64 samples per pixel limitation. If it could override this limit, the gain would greatly increase in the favor of our method.

8 Conclusion

We have introduced a set of three shaders able to represent at various scales the cumulated effects of the smaller scales without having to sample them, comprising the internal shadows, and taking the visibility into account. As all the required integrations are analytical, this provides at the same time efficiency and image quality (in particular, free of aliasing). However on the theoretical point of view, we would like to improve some of the approximations that have been done. Relaxing the low albedo hypothesis would be interesting either, e.g. using a polynomial law instead of a linear approximation.

The parameters of the shaders allow us to simulate various kind of pine-trees and fir-trees, and to modulate the characteristics inside a single tree (these modulations could be driven in time as well, e.g. to simulate the effects of the wind in a tree). We were able to derive these shaders because the objects we were interested in are very structured. Due to the extended use of the a priori knowledge, these three shaders can simulate nothing but trees made of needles. However, many objects in nature present one kind of structure or another, and even some similarities of structure, so it should be possible for each to derive shaders able to represent analytically each kind. The next step for us will be the simulation of other kind of trees, for which the structure is more stochastic (concerning the distribution and orientation of the leaves). Then it will be also interesting to handle larger scales, exploring larger structures than boughs inside and outside the trees...

Acknowledgments : We wish to thank Celine Loscos and Eugenia Montiel for re-reading this paper. Thanks to Pierre Poulin for discussing during this work.

References

[1] Barry G. Becker and Nelson L. Max. Smooth transitions between bump rendering algorithms. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 183–190, August 1993.

[2] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, volume 16(3), pages 21–29, July 1982.

[3] James F. Blinn. Models of light reflection for computer synthesized pictures. In James George, editor, *Computer Graphics (SIGGRAPH '77 Proceedings)*, volume 11(2), pages 192–198, July 1977.

[4] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12(3), pages 286–292, August 1978.

[5] Brian Cabral, Nelson Max, and Rebecca Springmeyer. Bidirectional reflection functions from surface bump maps. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21(4), pages 273–281, July 1987.

[6] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24, January 1982.

[7] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, May 1992.

[8] Dan B. Goldman. Fake fur rendering. *Proceedings of SIGGRAPH 97*, pages 127–134, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[9] Jay S. Gondek, Gary W. Meyer, and Jonathan G. Newman. Wavelength dependent reflectance functions. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, pages 213–220. ACM SIGGRAPH, July 1994.

[10] Xiao D. He, Kenneth E. Torrance, François X. Sillion, and Donald P. Greenberg. A comprehensive physical model for light reflection. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):175–186, July 1991. ISBN 0-201-56291-X. Held in Las Vegas, Nevada.

[11] James T. Kajiya. Anisotropic reflection models. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 15–21, July 1985.

[12] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23(3), pages 271–280, July 1989.

[13] Nelson Max. Hierarchical rendering of trees from precomputed multi-layer Z-buffers. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 165–174. Eurographics, Springer Wein, June 1996. ISBN 3-211-82883-4.

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

[15] Fabrice Neyret. Modeling animating and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), January–March 1998. ISSN 1077-2626.

[16] Tsukasa Noma. Bridging between surface rendering and volume rendering for multi-resolution display. In *6th Eurographics Workshop on Rendering*, pages 31–40, June 1995.

[17] Pierre Poulin and Alain Fournier. A model for anisotropic reflection. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24(4), pages 273–282, August 1990.

[18] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan. Developmental models of herbaceous plants for computer imagery purposes. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 141–150, August 1988.

[19] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graphics*, 2:91–108, April 1983.

[20] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 313–322, July 1985.

[21] Jos Stam. Stochastic rendering of density fields. In *Proceedings of Graphics Interface '94*, pages 51–58, Banff, Alberta, Canada, May 1994. Canadian Information Processing Society.

[22] Jos Stam and Eugene Fiume. A multiple-scale stochastic modelling primitive. *Graphics Interface '91*, pages 24–31, June 1991.

[23] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In Robert Cook, editor, *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 119–128, August 1995.

[24] Stephen H. Westin, James R. Arvo, and Kenneth E. Torrance. Predicting reflectance functions from complex surfaces. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):255–264, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

Interactive Rendering of Trees with Shading and Shadows

Alexandre Meyer Fabrice Neyret Pierre Poulin
iMAGIS-GRAVIR/IMAG-INRIA LIGUM

Abstract. Our goal is the interactive rendering of 3D trees covering a landscape, with shading and shadows consistent with the lighting conditions.
We propose a new hierarchical IBR representation, consisting of a hierarchy of Bidirectional Textures (sort of 6D lightfields). To improve the performance of shadow calculations, we associate to it a hierarchy of visibility cube-maps.
For a tree, the levels of the hierarchy may correspond to a small branch plus its leaves (or needles), a larger branch, and the entire tree. A Bidirectional Texture (BT) provides a billboard image of a shaded object for each pair of view direction and light direction. We associate a BT for each level of the hierarchy. When rendering, the appropriate level of detail is selected depending on the distance of the tree. The illumination reaching each level is evaluated using the visibility cube-map. Thus, we obtain very efficiently the shaded rendering of a tree with shadows without loosing details, contrary to mesh simplification methods. We produced a 7 to 20 fps animation of a scenery with 1,000 trees.

Keywords: Real-time rendering, natural scenes, forests, IBR, levels of detail, billboards

1 Introduction

Walk- and fly-throughs natural landscapes with the best possible visual quality have been a continuous challenge since the beginning of Computer Graphics. In real-time applications such as simulators and games, users want ever more convincing realism (*i.e.,* more trees, better looking trees). In off-line applications such as impact studies and special effects, users want the rendering softwares to compute ever faster.

The rendering of tree covered landscapes is especially demanding, because of the total amount of details, the complex lighting situation (shading of small shapes, shadows, sky illumination), and the inconvenient visibility situation (there are no large blockers that can be efficiently exploited).

Fortunately, trees also show interesting properties: their complexity of appearance lies on an intense redundancy of elements, and their structure is naturally hierarchical: trees are composed of a trunk and main branches, which group small branches together, whose boughs are made of leaves or needles; and leaves, needles, boughs, branches and even trees resemble to each others. These properties allows many modeling simplifications: elements of a family can be represented as instances of a few base models, and complex objects as a collection of similar simpler objects.

In this paper, we exploit further this notion by considering alternate and hierarchical representations and techniques to efficiently render trees, including shading and shadows. Our goal is to render with good image quality larger and larger landscapes covered with denser and denser forests of trees.

To achieve this level of efficiency and quality, we introduce a hierarchical bidirectional texture (HBT), inspired by recent image-based rendering (IBR) representations. It encodes the appearance of a tree model using several levels of detail relying on the hierarchy described above, for many view and illumination directions. This allows for fast rendering adapted to the amount of detail required.

The same hierarchical organization of a tree is also exploited for the visibility occlusion occuring within the elements of a hierarchical level. Combining the occlusions by traversing upwards the shadowing structure provides efficient and adapted soft shadows within the tree itself, and also between different trees. This allows for the real-time calculation of shadows while the light source is moving.

All these techniques have been implemented in hardware rendering under the current API of OpenGL, and results are provided and discussed.

This paper is organized as following. We review in Section 2 various approaches related to the rendering of natural scenes. We describe in Section 3 our representation, the *Hierarchy of Bidirectional Textures* (HBT). We explain in Section 4 how to render an object using it, then we provide in Section 5 the global algorithm for the rendering of the scene. Once the use of the representation is clear, we detail in Section 6 how to built it from an existing object (*e.g.,* a polygonal tree model). We describe our implementation, show our results, and give performance measures in Section 7, before concluding.

Contributions:

- The use of true Bidirectional Textures (BTF). The original paper (which scope is out of the CG field) and database [3, 2] decreases in practice the amount of degrees of freedom. Transparency is not considered either. In the CG field, the [4] representation encodes a normal to be used with Phong reflection model, instead of sampling the aspect when the light direction varies.
  Moreover, we store separate BTF for several light types, namely a quasi-point light source (the sun), and an ambient light source (the sky). This allows us to handle at rendering time (*i.e.,* in real-time) the separate changes in sky or sun color or intensity (such as evolving sunset or sky getting overcast), and multiple light sources (*e.g.,* helicopter headlights).
- Hierarchical Bidirectional Textures (HBT). We propose a complete level of detail representation based on BTF, which relies on the existing scene graph that describes objects like trees, taking advantage of massive use of instancing.
- Visibility Cubes-Maps (VCM) are not really a contribution, since they are a natural and simple way to encode the visibility around a location, which is done more precisely and in a more general scope by other representations such as [5, 26]. We use it for getting the shadows on trees in real-time, with a possibly moving light direction.
- Hierarchical Visibility Cubes-Maps (HVCM). They separate the occlusion between levels, *i.e.,* occlusions caused outside the object, and occlusions caused inside the object, outside a sub-object (Figure 2.2(bottom)). The total occlusion in one direction is obtained by combining the maps. In the context of massive instancing, this allows to factorize the maps corresponding to internal occlusion.

2 Previous Work

Despite the complexity of the task, flight simulators were an early application of Computer Graphics, running on dedicated machines. Trees and other objects (*e.g.,* buildings) were painted on the floor, represented by very simple polyhedrons (pyramids, cubes), or using the very first IBR representation, the sprite. Sprites have turned to billboards (another early IBR representation) living between 2D and 3D worlds, always facing the observer but attached to a 3D frame. These are still used in nowadays simulators [17] and off-line video productions with short schedule. Then games of the mid-eighties to

the mid-nineties, *i.e.,* before the arrival of 3D graphics boards, have extensively used sprites and billboards to produce interactive fake 3D environments, *e.g.,* for car races. With the generalization of hardware-accelerated 3D and textures (coming with the increase of computational power and available memory), workstations since the end of eighties and home PC since the end of nineties have started using cross-trees, made of 2 to 3 orthogonal faces textured with a transparent image, thus behaving like 3D objects. This has been generalized in applications up to a dozen of textured faces or more, figuring the foliage.

Several early papers have described methodologies to model the shape of trees. These are not in the target of our review, which only deal with their efficient rendering. Numerous methods have been progressively introduced to precompute polygonal levels of details or simplify meshes on the fly. These are also out of our scope, as they do not behave very well for trees, suffering numerous artifacts during a geometric simplification (nonetheless, an hybrid solution dedicated to tree [32] did nice work, but is still not sufficient for our quality and real-time goals).

The most interesting solutions have come with representations alternative to meshes, proposing a way to get efficiently both quality and quantity. An early one were particle systems [23, 24], consisting of replacing geometry by strokes, thus producing the first dense forest images. Although originally too slow, the approach is now used in real-time applications. Since 1995, the creativity in the field of alternate representations has exploded, following different tracks:

- IBR reuse real or synthetic views of objects [10, 27, 29, 8, 1], and in particular lightfields [9, 6], storing the color for the various possible rays hitting an object. While it can capture and redisplay most views of a tree, even under a different perspective view, the shading and shadowing remain fixed, and a lightfield occupies at least several Mb of memory. It has been applied to surfaces [16, 34] to incorporate some shading effects, however its memory requirements is still too large to represent good quality trees.
- On a dual way, textures have expended to reproduce the view angle dependency with birectional texture functions or relief textures [4, 20], or even to simulate objects in volume with layered impostors, volumetric textures and LDI [25, 15, 28]. Between these last techniques and the early cross-trees, Max *et al.* have combined in various ways Z-buffers used as a representation [13, 11, 12], with different orientations, organized in a level of detail hierarchy, or more recently with multilayers. Some of these methods have been use explicitly for trees: volumetric textures [15, 19], and of course the Max's work.
- A totally different approach lies in point-based rendering, such as surfels [21]. To date, it is not adapted to sparse geometry like foliage, but we think it might be a promising technique.

Bidirectional reflection distribution functions (BRDF) [18] encodes the proportion of reflected light given a pair of incident and reflected (view) directions. They are thus used in quality rendering, and represent precisely the aspect lacking in usual IBR and textural encodings. BTFs, and somehow surface lightfields, are a way to introduce this missing dimension.

Our representation shares various aspects with the BTFs [4, 3], the hierarchical IBR of Max *et al.* [12], billboards, and lightfields. Like Max *et al.* [12], we manage levels of detail using a hierarchy of alternate representations: an alternate-type object encodes a set of smaller objects, which can be themselves classical geometry or alternate-type.

Depending of the apparent size, we consider either the large object or the collection of smaller ones, and so on along the levels. Our alternate-type is a BTF, instead of layered Z-buffer as for Max *et al.* [11, 12]. These BTFs are precomputed using multiple local pre-renderings like in [4], but using the 6 degrees of freedom of [3, 2]: position on texture, view direction, and light direction. Indeed, we really sample all these directions because we want to encode 3D objects, while [3, 2] only sample only 3 of the 4 degrees of freedom (their purpose is the encoding of surface materials, not 3D objects. In case of anisotropy, they simply add a second samples set). As for lightfield and BRDF representations, we have to manage the sampled representation of a 4D direction space. However, our structure is much more compact despite its 6D nature, as we use it only for distant objects that appear small on the screen. Using billboards, we share the representation of 3D objects with textures which adapt to the point of view.

The auxiliary representation we introduce to accelerate the shadows was inspired by horizon maps [14, 31] and visibility precomputed for cells [5, 26], in a more approximative fashion. On the other hand it is 3D (all directions are valid) and hierarchical again, being connected to the HBT representation. We detail this representation in the next section.

3 Our Representation

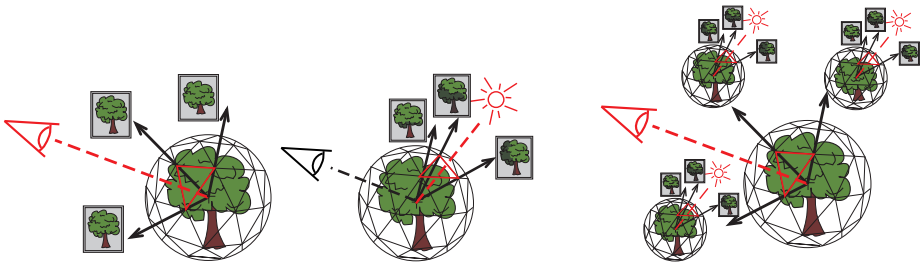
There are 3 elements in our representation:

- The **BTF encoding** of an object, an alternate representation that produces similar images of this object even when varying the observer and light locations. This is explained in Section 3.1.
- The **HBT structure** associated to a complex object, treated in Section 3.2, which extends the hierarchical scene graph of this object with level of detail information: a BTF is associated to each level. Reciprocally, most of the objects contained in a given level have an associated BTF. As this work focuses on trees, these objects are generally organized as multiple instances of one similar base object (*e.g.,* a branch), plus possibly one mesh object (*e.g.,* a piece of trunk). This greatly reduces the total amount of different BTFs (3 in our implementation).
- The structure for shadows, *i.e.,* the **hierarchy of visibility cube-maps**, that is described in Section 3.3.

3.1 Bidirectional Texture Functions

The principle of the representation is to associate a billboard representing an object with each pair of view and light directions (see Figure 1 and image 5), much like a BRDF associates a color with a pair of such directions (*i.e.,* we must manage 6D instead of 4D). If we limit ourselves to objects that are not pathologic shading-wise, only a small set of directions will be necessary. Moreover, these billboards are targeted to replace the detailed 3D representation of objects appearing small, and therefore their texture resolution can be very coarse (*e.g.,* 32×32 or 64×64 in our implementation). At rendering time, we reconstruct an image for given view and light directions by interpolating between the closest precomputed billboards. We detail the rendering in Section 4 and the BTF construction in Section 6.

It is better to avoid mixing in the same billboards the effects of sun illumination and ambient light, otherwise it would not be possible to tune separately their colors and intensities, or to consider several light sources. E.g., we would like to deal with the



1.1: A billboard is reconstructed from a given view direction by combining the 3 closest images stored in the sampled sphere of view directions.

1.2: A billboard is reconstructed from a given light direction by combining the 3 closest images stored in the sampled sphere of light directions.

1.3: The complete BTF allows to reconstruct a billboard for given view and light directions by combining up to 9 stored images (in our implementation), much like a sampled BRDF allows to reconstruct an energy value by combining a set of sampled values.

Fig. 1. The Bidirectional Texture Function representing a tree.

weather getting overcast, showing only the ambient term, the sun turning red while the sky remains light blue, or extra light sources such as helicopter headlights... Thus we store separately "ambient" billboards associated with each view direction. Note that this "ambient light" differs from classical Phong ambient as it takes occlusions into account.

In our implementation, we use either 6, 18, 66, or 258 samples for the view directions, evenly distributed on a sphere. For each sample direction, we associate a similar sampled sphere for light directions[1]. A small color texture with transparencies is stored for each pair of directions. To determine efficiently the closest 3 samples at rendering time, a precomputed table gives the 3 closest sampled directions for any given vector.

The instances of the obtained BTFs are associated with a local frame of reference and a scale factor, much like classical objects.

3.2 Hierarchy of BTFs (or HBTs)

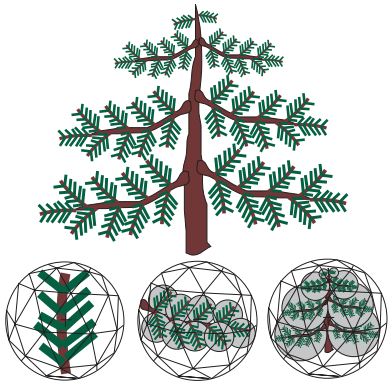
A BTF is associated to each level in the hierarchy (Figure 2.1), representing its coarse level of detail. A level consists in a set of objects. Each of them can be unique or an instance of a given base object, located and oriented using its frame of reference. Most base objects should have an associated BTF, excepted if their shape is very simple. At rendering time, either the level's BTF or the set of objects will be used, depending on the distance.
NB: in this paper, we call "highest" the coarsest level, representing the whole object.

3.3 Hierarchy of Visibility Cube-maps for Shadows

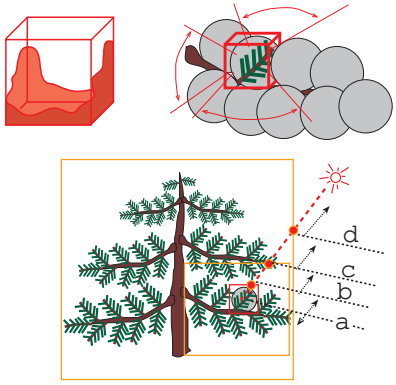
A single shadow evaluation is used for a given object (*i.e.,* for the current level of detail). As we are using hardware rendering we cannot rely on it to compute shadows, and because the scene is complex we cannot evaluate it on the fly using a shadow ray [2].

[1] The number of samples can be different on the 2 spheres. In practice, the quality of interpolation is more sensitive to the view directions than to the light directions.

[2] A global shadowmap cannot be used either because the objects are partly transparent.



2.1: Hierarchy of BTFs. *Top:* a typical tree model, showing a natural hierarchy (leaves or needles, small branches, main branches, trunk). *Bottom:* the 3 BTFs associated to the 3 levels of the hierarchy. For instance, the main branch (middle) is composed of instances of small branches (left).



2.2: Hierarchy of visibility cube-maps. *Top:* a visibility cube-map is associated to each instance, here a small branch on a main branch. It registers the occlusion (figured in red) generated by its environment in every direction. *Bottom:* to obtain the total visibility in a given direction, one has to combine the corresponding values in the cube-maps of the higher levels, *e.g.,* b, c, d for the small branch. Self-shadows (a) are already encoded in the branch BTF.

Thus we want to do precomputations, but we still want the light direction to change in real time.

Note that self-shadows are already represented in the BTF associated with the object, thus we only have to account for shadowing caused by occluders outside the object (*i.e.,* the current level). We store the visibility around the current object (Figure 2.2(top)), using cube-maps. The idea is analog to the horizon maps [14, 31] (apart that these are 1D while we need 2D), or to the visibility precomputed per cell [5, 26]. A cube-map represents the visibility at a given location. For each light direction, we precompute the total amount of light reaching the location, which gives the amount of occlusion in this direction, stored in the corresponding pixel of the visibility cube-map.[3] For an object, we sample the visibility at several locations (in our implementation, it is at the 8 corners of the bounding volume), and the visibility at any location is estimated by interpolating the cube-maps. In the following, the "visibility cube-map" associated to an object refers to this set of cube-maps, and its value for a given direction refers to the interpolation of the corresponding value in the cube-maps within the set. The pixel values correspond to opacities (in the range [0,1]), and are used at rendering time to modulate the color of the billboard in order to get the shadows. Note that the dynamic of this occlusion value allows for transparent objects, antialiasing and soft shadows.

Computing and storing a different cube-map for every instance of objects in our scene would be excessively costly (cf. the total amount of small branches in the forest). Instead, we rely on a hierarchical structure following the HBT hierarchy, and separate the occlusions occuring inside one level to the occlusions occuring outside: for a given level in the hierarchy, a visibility cube-map is associated to each instance of the objects in the set. It only represents the occlusions caused by the objects within the set. At rendering time, the illumination received by an object from a given light direction is modulated by combining the values of the cube-maps at the current and higher levels of the hierarchy, as shown on Figure 2.2(bottom).

[3]The visibility cube-map is thus the dual of a shadow map, which encodes the visibility of every location in the scene for one single light direction.

The status of shadows is thus as follows:

- self-shadowing is encoded in the BTF (figured by a in Figure 2.2);
- a visibility cube-map encodes the blocking of light between its associated object and the cube-map associated to the whole hierarchy level (corresponding to b and c in Figure 2.2);
- the cube-map at the highest level of the hierarchy encodes the blocking of light outside, within the whole scene (stored in d in Figure 2.2).

Therefore a different cube-map has to be stored for every instance of highest level objects in the scene. Oppositely, for an intermediate hierarchy level, as the cube-maps of objects handle only the visibility within this level (and not relatively to the entire scene), we only need one cube-map per instance of an object in the level. For instance, each main branch constituting a tree has its own cube-map, but as the trees in the landscape are instances of one single tree, we do not need to consider a new visibility cube-map for each branch in the scene. In our implementation, the maps are $32 \times 32 \times 6$, and the highest level corresponds to a tree, so the total required memory is acceptable (see Section 7). If necessary, we could add an extra level consisting of a group of trees.

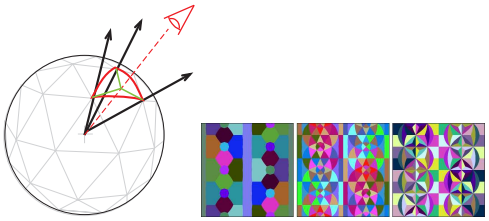
We also precompute the average value of each cube-map, that gives the ambient visibility (*i.e.,* the sky occlusion); this will be used to evaluate the ambient illumination.

4 Rendering an HBT

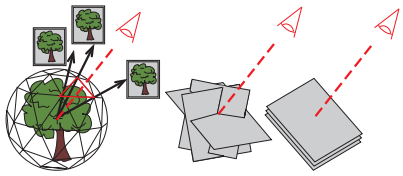
To render an HBT, first we need to determine the appropriate level of detail according to its apparent size in the image. Then all the objects in the 'opened' levels of the hierarchy, either polygonal meshes or BTF, are rendered from back to front. To render a BTF, we reconstruct a billboard for the current observer and light directions, and then compute its illumination taking into account light color, sky color and shadows.

4.1 Reconstructing an Image from a BTF

Formally, the reconstruction problem is mainly the same than for a BRDF: we have to sum weighted values of the closest samples. 3 classical issues are then: how to choose the best samples and weights, how many samples, and how to find them efficiently.



3.1: *Left:* Given the observer direction, the 3 closest sampled view directions must be found, and their 'distance' to the current direction calculated. *Right:* The 3 tables giving the precalculated 3 closest sampled directions (the colors figure the sample ids).



3.2: The 2 projection strategies concerning the 3 images corresponding to the closest view directions: cross-tree like (middle), and billboard-like (right).

Our implementation is quite naive regarding the classical issues. As we separate the 4D table into a sampled sphere for view directions and another one for light directions, we simply have to find in which cell bounded by 3 samples a given direction falls (Figure 3.1(left)). To be more efficient, we construct a precomputed table that directly

provides the 3 sample directions for any given direction (Figure 3.1(right)). The blending coefficients correspond to the barycentric coordinates within the cell on the sphere. As the 3 billboards corresponding to the 3 samples from the view direction result themselves from the blend of the 3 samples from the light direction, we get 9 billboards to blend. The blending coefficients are the product of the barycentric coordinates relative to the view cell and to the light cell. However 9 can be considered a bit high[4]. To reduce this number, we do not consider samples associated to coefficients below a given threshold (and we renormalize the other coefficients). It might be possible to reduce even more this number using some similarity criteria, considering only the samples that add enough information.

As we deal with images instead of scalars in BRDFs, there are 2 extra issues: how to project each image on screen, and how to blend them together, knowing that we want to use the hardware Z-buffer that is not totally compliant with transparent images.

Two main choices exist to address the projection on screen, depending whether the billboard or the cross-textured philosophy is preferred for trees (Figure 3.2):

- considering each image as a projected slice, that should then keep its original orientation;
- considering each image as a billboard, that are consequently mapped at the same location, *i.e.,* facing the eye.

The first solution provides more parallax effects especially if very few directions are available. The drawback is that the bounding volume depends on the angle of view, and that the opacity is poorly distributed (denser where the images superimpose). The second solution is well adapted for objects having a cylindrical or a spherical symmetry: as the image is always facing the observer, the bounding volume is more stable and the opacity better distributed. Note that the more the sampled directions, the closer the two solutions, so it is better to rely on the simplest one, *i.e.,* billboards. For very small sampling (*e.g.,* 6 directions), it is probably better to choose the strategy depending on the look of objects: in such case, we used the classical cross-images.

The last issue is the blending. The images have to be combined by weighted summation, and not blended together by alpha compositing, which would depend on the order of operations. Oppositely, we need to blend (by compositing) the result with the background image on screen.

$C = \sum_{i,j} \alpha_i \beta_j \, \mathtt{BTF}_a(i,j) \, \mathtt{BTF}_{rgb}(i,j)$,

$A = \sum_{i,j} \alpha_i \beta_j \, \mathtt{BTF}_a(i,j) \;\; = \;\; \sum_i \alpha_i \mathtt{BTF}_a(i)$,

$C_{pixel} = C + (1 - A) C_{pixel}$,

where α_i and β_j are the barycentric coordinates of the view and light directions.

Another problem is that we do not want the depth test of the Z-buffer to reject fragments that might appear behind a transparent fragment, especially when drawing intersecting billboards. A costly solution would be to process the reconstruction in a separate buffer (without Z-test), and to use the result as a texture to be blended with the image on screen. To simulate this process, although keeping the pipe-line compliance, we first darken the background according to the cumulated transparency of the billboards. As the same transparency occurs for the 3 images corresponding to one view direction sample, we only have 3 transparency maps to consider. "Darkening" is done using the `ADD` blending equation of the OpenGL API with `(0,1-SRC_ALPHA)` blending coefficients. The weight α_i is encoded in the alpha of the underlying polygon.

[4]Especially when noticing that considering directly a 4D-sphere for directions would have yield 5-vertex cells, *i.e.,* the shape of a 4D-simplex.

Then we draw the weighted textures (the polygon alpha is set to $\alpha_i\beta_j$), using `(SRC_ALPHA,1)` blending coefficients. Thus we have a total of 12 passes if all the billboards are to be drawn. To avoid Z-interactions, the Z values are stored only in the last pass[5].

4.2 Shadowing an Object

An object has to be modulated by the received illumination before it is drawn. The amount of light is simply obtained by combining the opacity values for the current light source direction in the visibility cube-maps associated to the current object and all the levels above (Figure 2.2). This value is multiplied by the light source intensity and color, then used as the polygon color when drawing the billboards as described above. We can also consider a cloud map to modulate the sun intensity.

$illumination_L = C_L \prod_{level\ k} 1 - \texttt{VCM}_k(L)$

To handle multiple light sources, we iterate the process above, yielding a maximum of 9 new passes per light source with our implementation. We could potentially exploit the fact that any additional light source in a landscape should have a more limited visual impact on the entire forest (*e.g.,* helicopter headlights) and therefore treat it only for the closer trees.

The last component is the ambient term: to consider diffuse illumination not directly due to the sun, namely, sky illumination, we have to sum another image corresponding to the ambient contribution. Since it does not depend on light direction, we only have to combine 3 billboards, associated to the view direction sampled sphere. The "ambient shadowing" is obtained by multiplying the average visibility associated to the cube-maps and the average cloud transparency.

$illumination_{amb} = C_{sky} \prod_{level\ k} 1 - average_\texttt{VCM}_k$

So $C = \sum_{light\ s} \texttt{HBT}(V, L_s)\ illumination_L + \texttt{HBT}_{amb}(V)\ illumination_{amb}$ where V is the view direction and L_s the direction of the light source s.

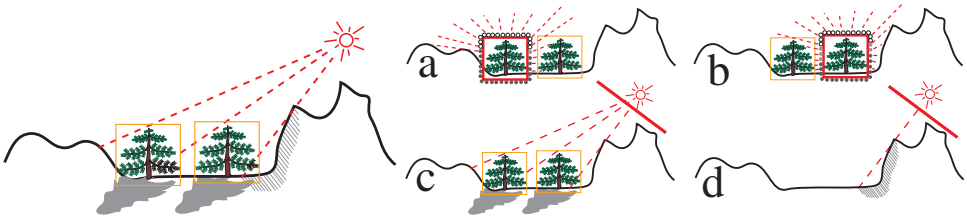


Fig. 4. The 4 kinds of shadow interaction between trees and terrain. *Left*: final aspect. *a,b*: The visibility cube-maps of each tree take into account the presence of both the mountain (and ground) and the other trees (self-shadows are included in the BTFs and in the cube-maps down in the hierarchy, see Figure 2.2). *c*: The alpha-shadowmap accounts for the soft shadows of the trees on the floor. *d*: The Z-shadowmap (or *depth map*) accounts for the self-shadows of the mountain (any of the numerous shadowing algorithms for terrain could be used instead).

[5] As the objects are sorted, storing the Z values of trees is not very useful anyway, as long as possible extra opaque objects are drawn first (the Z-test remains enabled).

5 Rendering the Landscape

The global scene rendering algorithm is simple: first we render the terrain (and possible extra opaque objects), then we consider all the objects at their highest level (*i.e.,* trees) from back to front, culling those that are clearly not visible (outside the view frustum, and if possible behind the mountains, using a conservative occlusion technique [35, 5, 26], or extending techniques dedicated to terrains [14, 30, 31]). The rendering of HBT objects is then processed as described in Section 4.

The last issue is the shading and shadowing of the terrain. We could use the same principle described in Section 4.2, applied at the vertices of the terrain mesh. But the resolution of the shadows would probably be too coarse, and a visibility cube-map would have to be stored at each vertex which is probably too much. We prefer to simply use shadowmaps [33, 7], recomputed when the light direction changes[6]. In fact, two different kinds of shadowmaps have to be combined: one dealing with transparent objects (the trees), and the other with opaque objects (the terrain), as illustrated on Figure 4(bottom). These shadowmaps are built by rendering the scene from the sun location. For the first shadowmap, only the trees transparency is drawn, as a grey level. The terrain is drawn invisible (*i.e.,* white): it should not produce a shadow, but it should hide the trees behind the mountains. For the second shadowmap, which takes into account the terrain self-shadowing (*e.g.,* the shadow of a mountain on the valley), a depth map is used (any terrain shadowing technique such as horizon map [14, 30] could be used instead). Note that SGI IR hardware provides this feature.

6 Building the Data

Our scenes are mainly composed of a terrain and many instances of one or more tree models. We need to encode an existing tree model in our representation, as transparently as possible for the user. Our only assumption is that instances are massively used to design a tree: *e.g.,* a tree is composed of numerous instances of a few branch models, which are composed of numerous instances of small branches and leaves (or needles). We rely on this hierarchy to organize our levels of detail.

We have two hierarchical structures to built: the BTFs encoding the appearance of each subset of a tree (including the tree itself), and the visibility cube-maps encoding how the environment modulates the received illumination of each element.

6.1 Building an HBT

A BTF is a collection of images of a shaded object from various view directions and light directions (Figure 5). We proceed in the same spirit than [4], adding the loop for light directions. We have two kinds of object:

- The bottom level of the hierarchy is composed of classical objects. We use a ray-tracer to produce the views of mesh models despite the final rendering is a Z-buffer. It allows us to easily get self-shadowing, antialiasing, transparencies, and to use complex shaders (*e.g.,* the shading of needles can be analytically integrated, while the cylinders should be discretized if a Z-buffer were used).
- The higher levels of the hierarchy are mainly composed of BTFs, plus possibly a few simple polygonal meshes (*e.g.,* for the trunk). We use a Z-buffer to produce the views of these higher levels, in order to guarantee a consistent shading

[6]Here a shadowmap can be used because the terrain is opaque (while trees are not).

whether the lower or the higher level of detail is used during the interactive rendering. Indeed, we use the same rendering algorithm than used for interactive rendering, which provides us with shadows casted between the elements thanks to the visibility cube-maps structure.

In addition to the light-dependent BTFs, we also have to build the small set of light-independent BTFs corresponding to the ambient term (used for the sky illumination). They are currently produced by rendering the object with only the ambient term in the shading, taking into account the ambient visibility. In our implementation, the views are 32×32 or 64×64 RGBA images.

6.2 Building the Visibility Hierarchy

A visibility cube-map associated to an object encodes the light occlusion in every direction due to the environment, but limited to the elements in the hierarchical level it belongs to (or the entire scene for the highest level).

Thus for each pixel of the cube-map, the energy transfer between the light source (the sun) in the corresponding direction and the bounding volume of the object (taking occlusions into account) should be evaluated, and the amount of occlusion should be stored. As a cube-map has to be computed for each instance of a tree model in the scene, taking into account a complex environment, this precomputation can be costly.

We implemented an approximative, yet very efficient simplification, using our OpenGL-based rendering algorithm. We place the camera at the center of the cube, and render the appropriate environment (in fact, only its transparency, as a grey level) in the 6 directions. This corresponds to the illumination received at the center, instead of the average value received by the object.

Once the cube-map is built, we need to compute its average value, that is used to evaluate the sky illumination. Note that the cube-map is a generalization of the horizon map [14], while the average value is equivalent to an accessibility (or exposure) map.

7 Results

In our implementation, billboard images are 32×32 or 64×64 RGBA textures. Observer and light directions spheres are discretized with 6 or 18 evenly distributed samples. Thus regular BTFs are $32 \times 32 \times 4 \times 6 \times 6$ (144 Kb), $32 \times 32 \times 4 \times 18 \times 6$ (432 Kb), or $32 \times 32 \times 4 \times 18 \times 18$ (1.3 Mb), and ambient BTFs are $32 \times 32 \times 4 \times 6$ (24 Kb) or $32 \times 32 \times 4 \times 18$ (72 Kb). Our tree models are produced by an L-system [22]; we used for our tests 2 kinds of pine-trees and a prunus tree. We consider 3 hierarchical levels: a small branch with leaves or needles, a main branch made of instances of small branches, and a tree made of main branches. The pine-tree has 30 main branches, 300 boughs, and 30,000 needles. The whole HBT thus consists of 3 BTFs (0.5 to 4 Mb), plus the same scene graph than the geometrical tree.

The scene contains 1,000 trees, so there are $8 \times (1,000 + 30 + 10)$ visibility cube-maps. The cubes are $32 \times 32 \times 6$ luminance textures (6 Kb), thus the visibility structure represents roughly $8 \times 6 \times 1,000$ Kb = 48 Mb. If this should be considered as too much[7], one could easily introduce a fourth hierarchical level consisting of a dozen of trees, which would divided the memory expense by the same amount.

Precalculation time is about 75 minutes, 2/3 for visibility and 1/3 for the HBT.

[7] However, note that this data structure lies in main memory, and is never loaded on the graphics board.

Our global rendering algorithm is quite naive: we cull only the trees outside the view frustum, we sort the trees, then the parts of the trees for 'opened' levels of detail. We consider sun and sky illumination (*i.e.,* one light source plus the ambient). Our billboards combining optimization requires on average 5 images instead of 9. In the worst case we have to draw 15 billboards per object [8], which requires a high fill rate. The SGI we use has no multitexturing ability. However recent graphics boards such as Nvidia or ATI do have this feature, which allows to decrease the number of passes down to 5 (or 3, if the billboards of low contribution are eliminated).

We produced a 640×480 animation, using a $Onyx^2\ IR$, showing a fly over a scenery covered by 1,000 trees. The frame rate is 7 to 20 fps. We observed a 20% gain using an Nvidia board on PC, without using multitexturing. Using this feature, we should gain another factor of 2 to 3.

A pine tree represented by a polygonal mesh is about 120k polygons. Using the lowest level of detail (*i.e.,* maximum resolution) yields a gain of 8 in the rendering time. Switching to the middle level gives an extra gain of 18. With the highest (*i.e.,* coarsest) level, we have an other gain of 30.

8 Conclusion

We have introduced a new IBR representation for trees, which allows for very efficient quality rendering with complex effects such as shading, shadows, sky illumination, sun occlusion by clouds, and motion of the sun. Our naive implementation runs at 7 to 20 fps on an $Onyx^2\ IR$ on a scenery containing $1,000$ trees (it could be 2 to 3 times faster using a graphics board with multitexturing ability).

The HBT representation consists of a hierarchy of bidirectional textures, which provides a billboard for each given observer and light directions, plus a visibility hierarchical structure for the shadows (self and casted). Despite the 6 degrees of freedom of the BTF, the required memory of a few Mb is acceptable.

Specularities, BRDF and transmission are also handled by our method, but precision is limited by the sampling density of view and light directions. Sampling directions or cube-maps locations potentially yield artifacts, as linear interpolation does not reconstruct perfectly the data. This issue disappears with higher sampling, at the price of memory. Thus there is a trade-off between visual quality vs memory used. However, results show that nice real-time fly-over can be produced with reasonable memory, even when using several kinds of trees.

A far more efficient global rendering algorithm could be implemented, using grids and smart culling methods, thus allowing the interactive visualization of much larger landscapes. Various different implementation choices could be made on the HBT rendering and construction. For instance, a more exact visibility evaluation could be precomputed.

It would be interesting to see how the future graphics boards will include features easing the rendering process: SGI *Infinite Reality* implement 4D textures; Nvidia chips should do the same soon. Once 4D textures will be treated like any texture interpolation, filtering and compression wise, lightfields will be directly available in hardware. If 6D textures are also managed some day, BTFs will be directly available as well...

[8] 3 for darkening the background, 9 for the direct illumination, 3 for the ambient (*i.e.,* sky) illumination.

References

[1] S.E. Chen. Quicktime VR - an image-based approach to virtual environment navigation. In *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 29–38, August 1995.

[2] K. Dana, B. van Ginneken, S. Nayar, and J. Koenderink. Columbia-utrecht reflectance and texture database, http://www.cs.columbia.edu/cave/curet/.index.html.

[3] K.J. Dana, B. van Ginneken, S.K. Nayar, and J.J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics*, 18(1):1–34, January 1999.

[4] J.-M. Dischler. Efficiently rendering macrogeometric surface structures using bi-directional texture functions. In *Rendering Techniques '98*, Eurographics, pages 169–180, 1998.

[5] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. In *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 239–248, 2000.

[6] S.J. Gortler, R. Grzeszczuk, R. Szeliski, and M.F. Cohen. The lumigraph. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 43–54, August 1996.

[7] A. LeBlanc, R. Turner, and D. Thalmann. Rendering hair using pixel blending and shadow buffers. *Journal of Visualization and Computer Animation 2(3)*, pages 92–97, 1991.

[8] J. Lengyel and J. Snyder. Rendering with coherent layers. *Proceedings of SIGGRAPH 97*, pages 233–242, August 1997.

[9] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 31–42, August 1996.

[10] P.W.C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. *1995 Symposium on Interactive 3D Graphics*, pages 95–102, April 1995.

[11] N. Max. Hierarchical rendering of trees from precomputed multi-layer Z-buffers. In *Eurographics Rendering Workshop 1996*, pages 165–174, June 1996.

[12] N. Max, O. Deussen, and B. Keating. Hierarchical image-based rendering using texture mapping hardware. In *Rendering Techniques '99*, Eurographics, pages 57–62, 1999.

[13] N. Max and K. Ohsaki. Rendering trees from precomputed Z-buffer views. In *Eurographics Rendering Workshop 1995*, June 1995.

[14] N.L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4(2):109–117, July 1988.

[15] A. Meyer and F. Neyret. Interactive volumetric textures. In *Eurographics Rendering Workshop 1998*, pages 157–168, July 1998.

[16] G.S.P. Miller, S. Rubin, and D. Ponceleon. Lazy decompression of surface light fields for precomputed global illumination. *Eurographics Rendering Workshop 1998*, pages 281–292, June 1998.

[17] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. 1993.

[18] F.E. Nicodemus, J.C. Richmond, J.J. Hsia, I.W. Ginsberg, and T. Limperis. Geometric considerations and nomenclature for reflectance. October 1977.

[19] T. Noma. Bridging between surface rendering and volume rendering for multi-resolution display. In *6th Eurographics Workshop on Rendering*, pages 31–40, June 1995.

[20] M.M. Oliveira, G. Bishop, and D. McAllister. Relief texture mapping. *Proceedings of SIGGRAPH 2000*, pages 359–368, July 2000.

[21] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. *Proceedings of SIGGRAPH 2000*, pages 335–342, July 2000.

[22] P. Prusinkiewicz, A. Lindenmayer, and J. Hanan. Developmental models of herbaceous plants for computer imagery purposes. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 141–150, August 1988.

[23] W.T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graphics*, 2:91–108, April 1983.

[24] W.T. Reeves and R. Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 313–322, July 1985.

[25] G. Schaufler. Per-object image warping with layered impostors. In *Rendering Techniques '98*, Eurographics, pages 145–156, 1998.

[26] G. Schaufler, J. Dorsey, X. Decoret, and F.X. Sillion. Conservative volumetric visibility with occluder fusion. In *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 229–238, 2000.

[27] G. Schaufler and W. Stürzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–236, August 1996. ISSN 1067-7055.

[28] J. Shade, S.J. Gortler, L. He, and R. Szeliski. Layered depth images. *Proceedings of SIGGRAPH 98*, pages 231–242, July 1998.

[29] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. *Proceedings of SIGGRAPH 96*, pages 75–82, August 1996.

[30] A.J. Stewart. Hierarchical visibility in terrains. In *Eurographics Rendering Workshop*, June 1997.

[31] A.J. Stewart. Fast horizon computation at all points of a terrain with visibility and shading applications. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):82–93, March 1998.

[32] J. Weber and J. Penn. Creation and rendering of realistic trees. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 119–128, August 1995.

[33] L. Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12(3), pages 270–274, August 1978.

[34] D.N. Wood, D.I. Azuma, K. Aldinger, B. Curless, T. Duchamp, D.H. Salesin, and W. Stuetzle. Surface light fields for 3D photography. In *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 287–296, 2000.

[35] H. Zhang, D. Manocha, T. Hudson, and K.E. Hoff III. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 77–88, August 1997.

Fig. 5. The BTF associated to the highest level (horizontal axis: 18 view directions, vertical axis: 6 light directions, 64×64 resolution per billboard, with colors and transparencies).

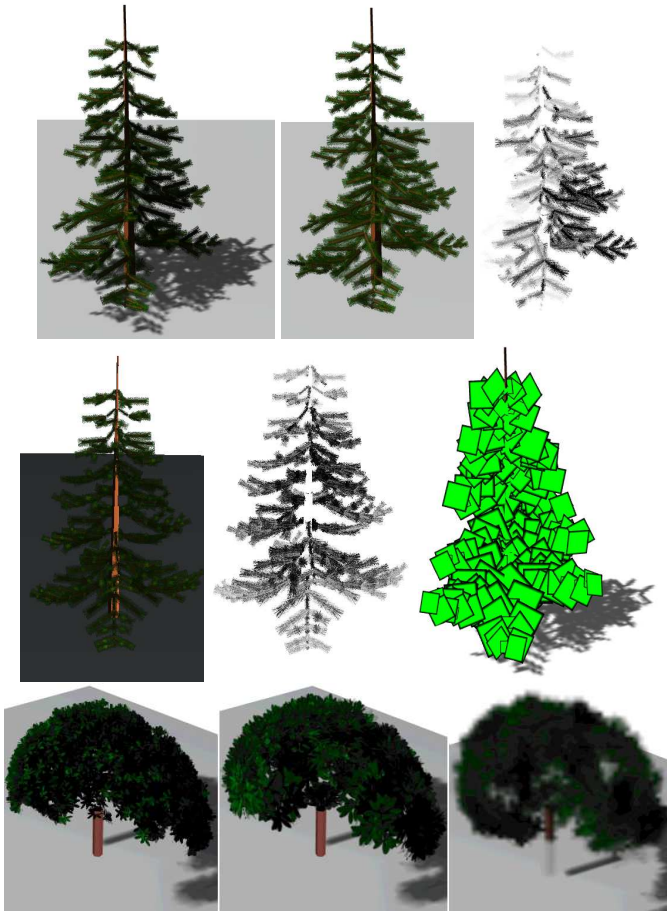


Fig. 6. *Top*: a pine tree with shadows, without shadows; amount of shadow. *Middle*: the pine with only ambient, the ambient visibility coefficient; the billboards used at the lowest level of detail. *Bottom*: a prunus tree drawn at 3 levels of detail.

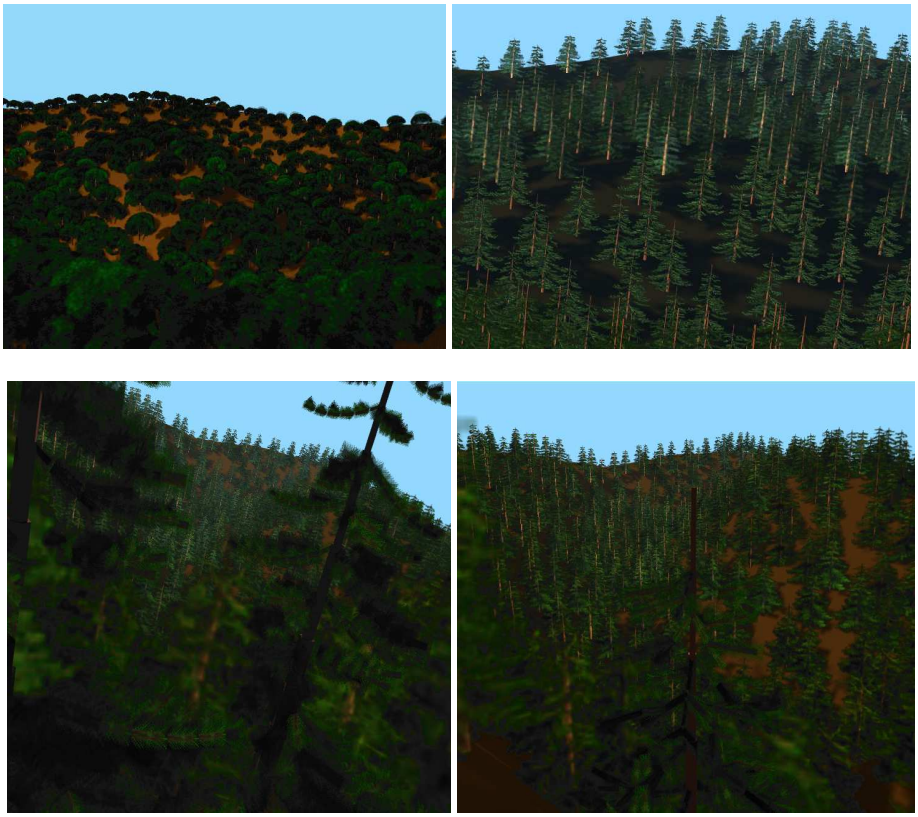


Fig. 7. 4 views in a forest (1,000 trees on a landscape, with shading, shadows and fog). The frame rate is 7 to 20 fps on our test machine. Note the detailed trees on the foreground.