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.

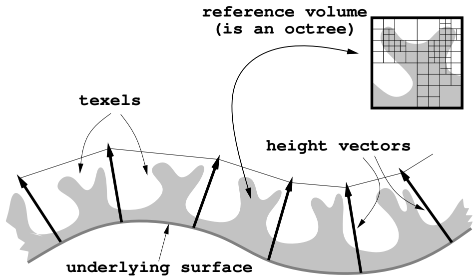

Fig. 1. Volumetric texture specification (cross-section).

- 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 $A = 255$.

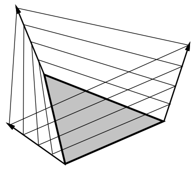

Fig. 2. A texel is drawn using extruded textured triangles.

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.

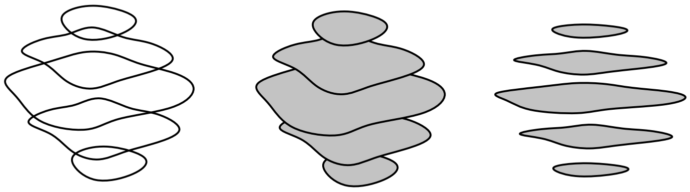

Fig. 4. *Left*: Contours. *Middle*: Filled contours. *Right*: Gaps appearing between slices at grazing view angles.

[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.

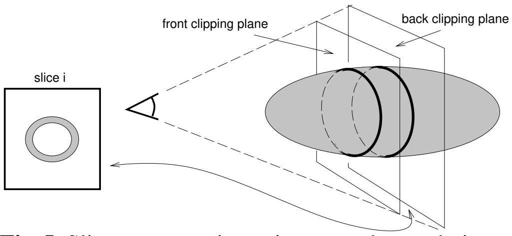

Fig. 5. Slices construction using a regular rendering tool.

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.

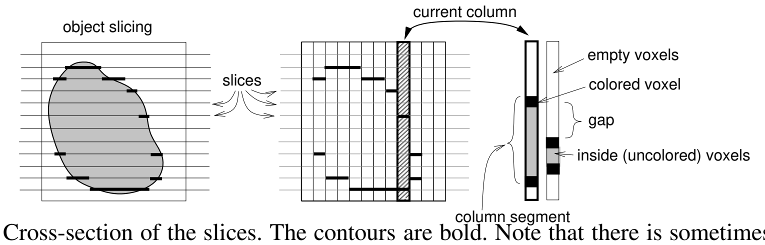

Fig. 6. Cross-section of the slices. The contours are bold. Note that there is sometimes a gap of several slices. This means that the intermediate slices have unclosed contours at this location.

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.

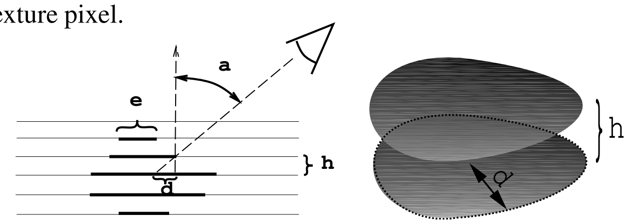

Fig. 7. *Left*: Slicing characteristics. *Right*: Limit for the second criterion: half of the inside of the narrowest slice is visible.

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).

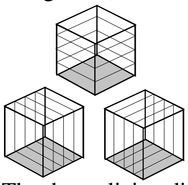

Fig. 8. The three slicing directions.

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).

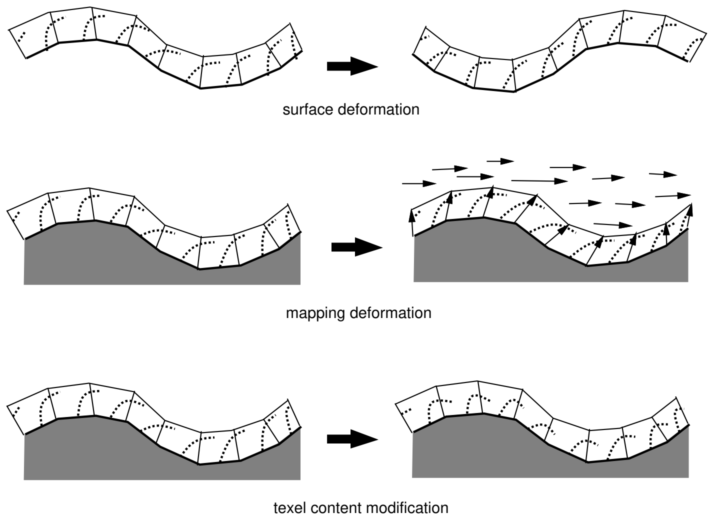

Fig. 9. The three modes of animation, that also correspond to three scales.

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

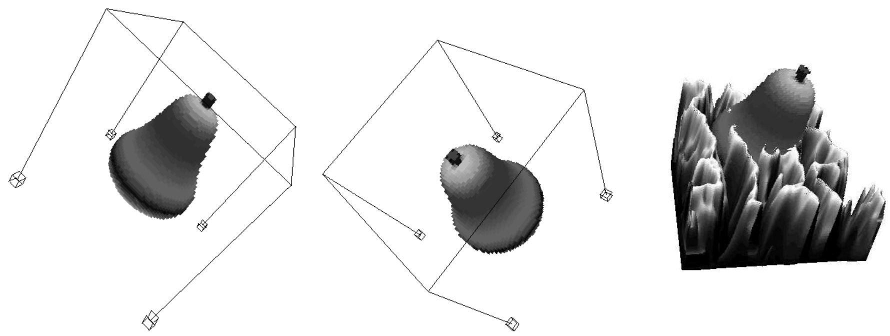

Fig. 10. A single pear texel at resolution 64^3. *Right*: Surperimposition of two texels.

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.

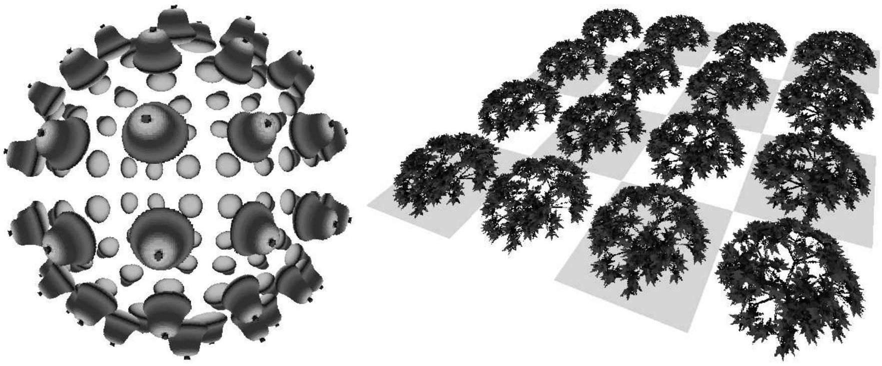

Fig. 11. Left: Mapping of 96 pear texels on a sphere. **Right:** Mapping of 16 bushes.

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.

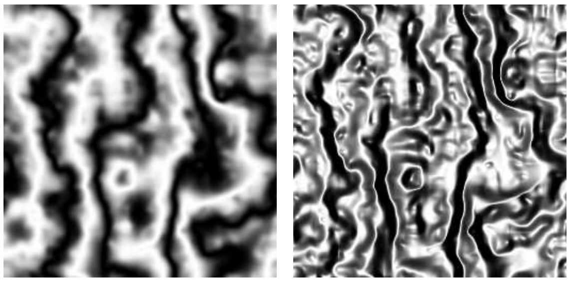

Fig. 12. Cyclical Perlin noise used to generate the height field, and the illumination computed from its gradient.

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

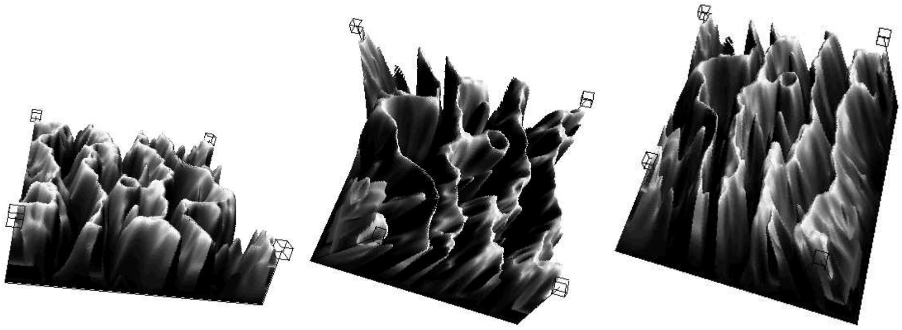

Fig. 13. *Left*: A texel at resolution $256 \times 256 \times 64$ created from the height field. *Middle and right*: Deformation of the texel by modifying the height vectors.

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)).

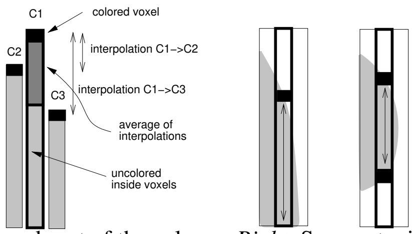

Fig. 14. Filling of the exposed part of the column. *Right*: Segment with one or two colored ends.