INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Perlin Textures in Real Time using OpenGL

Antoine Miné Fabrice Neyret
iMAGIS-IMAG, bat C
BP 53, 38041 Grenoble Cedex 9, FRANCE
Fabrice.Neyret@imag.fr

http://www-imagis.imag.fr/Membres/Fabrice.Neyret/

No 3713

juin 1999

THÈME 3

Rapport de recherche

ISSN 0249-6399

Perlin Textures in Real Time using OpenGL

Antoine Miné   Fabrice Neyret
iMAGIS-IMAG, bat C
BP 53, 38041 Grenoble Cedex 9, FRANCE
Fabrice.Neyret@imag.fr
http://www-imagis.imag.fr/Membres/Fabrice.Neyret/

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet iMAGIS

Rapport de recherche n°3713 — juin 1999 — 18 pages

Abstract: Perlin's procedural solid textures provide for high quality rendering of surface appearance like marble, wood or rock. This method does not suffer many of the flaws that are associated with classical image mapped textures methods, such as distortion, memory size, bad continuity through objects. Being based on a per-pixel calculation, they were however limited up to now to non-real-time quality rendering as is ray-tracing. In this paper, we propose a way to implement Perlin texture using a real-time graphics library like OpenGL.

Key-words: image synthesis, virtual reality, procedural texture, Perlin noise.

(Résumé : tsvp)

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Textures de Perlin en temps-réel avec OpenGL

Résumé : Les textures procédurales pleines de Perlin permettent un rendu de qualité pour des surfaces comme le marbre, le bois ou la pierre. Cette méthode ne souffre pas de la plupart des problèmes que rencontrent les méthodes de plaquage de texture classique, comme les distortions, l'occupation mémoire, la mauvaise continuité d'une composante géométrique à l'autre. S'appuyant sur un calcul par pixel, elles étaient toutefois limitées jusqu'à maintenant aux rendus de qualité non temps-réel, comme le lancer de rayon. Dans ce papier, nous proposons une méthode pour qui permet d'implémenter les textures de Perlin en s'appuyant sur une librairie graphique temps-réel comme OpenGL.

Mots-clé : synthèse d'images, réalité virtuelle, textures procédurales, bruit de Perlin.

1 Introduction

Perlin's procedural solid textures are often used to generate complex looking surface appearance such as marble, wood or rock. They have numerous interesting properties, compared to classical image textures:

- They are computed in 3D, not on the surface, which avoids surface parameterization problems that usually produce large distortions on image mapped textures.
- For the same reason, texture features like a vein in marble can easily continue from one element of a composed object to the other, while using classical textures a mapping continuous and coherent through objects have to be found (which is uneasy).
- Almost no memory is used, as the texture values are computed on the fly.
- The resolution is adaptive, each iteration adds increasingly small details and can be stopped once the pixel size is reach. Having a classical image texture both covering a whole surface and having very fine details (to allows for close point of views) can need a lot of memory (e.g. $10,000 \times 10,000$ resolution).
- As they are procedural, no redundant design work is necessary, and no repetition appears. The artist rather controls high level parameters, such as the size of perturbation, the amount of turbulence, the kind of patterns, their size, orientation and location, the range of colors, etc.

However Perlin's textures are based on a per-pixel calculation, thus they cannot be computed in real time, but rather used in a realistic rendering algorithm like ray-tracing. As a consequence, real-time graphics library such as OpenGL only knows image mapped textures.

It would be very interesting to get the quality and ease of Perlin's textures, with the interactive rendering rate of real-time graphics libraries like OpenGL. This would allow for high quality images in a real-time application. This would also provide a large acceleration to non-real-time quality rendering.

Using the numerous features of extended OpenGL such as 3D texture coordinates, multipass, color matrix and look-up tables, we demonstrate in this paper that the Perlin's noise equation can be translated in terms of per-polygon mapped texture rendering.

2 Previous Work

2.1 Perlin's textures

Perlin introduced his model in 1985 [Per85]. Since then it has largely been used in all the existing high quality image synthesis platforms such as Maya, Explore, PowerAnimator or Softimage.

This model contains two ideas:

- It is a *procedural* texture model, which means that the value at a point results from an on the fly calculation.
- It is a *solid* texture model[1], which means that the appearance on the surface reveals data that are defined in volume, as if the object was sculpted in a block of material.

The procedural model is based on *turbulent noise*, that is a continuous self-similar function providing fractal looking patterns. This fractal noise $t(x)$ is defined in 1D as the fractal sum of a simple noise $s(x)$: $t(x) = \frac{\sum_0^n \frac{1}{2^i}.s(2^i.x)}{\sum_0^n \frac{1}{2^i}}$ A value of 4 for n generally gives good results, but one can let the sum add details up to the pixel size. Moreover, a lower value can be used to get smooth patterns. The $s(x)$ *noise function* is both continuous and random, and has a pseudo-period that can be controlled. It is built by interpolating smoothly random values defined on the nodes of a grid. Affine transformations of x allows for controlling the size of patterns (i.e. the frequency of the lowest component), and there location (i.e. the position of a pick in the values relative to a geometric feature). Formulas are identical in 3D, taking x as the vector $\vec{x} = (x, y, z)$. $s(\vec{x})$ is thus a function from $\mathbb{R}^3$ to $\mathbb{R}$, which smoothly interpolate the values given on a 3D grid.

In fact no 3D grid really need to be built, neither infinite array: hashing techniques [Arv90, EMP$^+$98] allows for the simulation of uncorrelated data using a simple small 1D grid of precomputed random values. The hashing of the indices greater than the array size allows for the generation of uncorrelated sequences: e.g.

[1] also called 3D textures, which should not be confused with *volumetric textures* [KK89] that really design 3D expanded objects.

the sequences $\{i\}_i, \{\alpha.i\}_i, \{\beta+i\}_i$ where the parameters α and β are large and generally primes, are uncorrelated once the indices are hashed. A permutation function $\sigma(i)$ of the indices allows for the production of uncorrelated components in multidimensional data. e.g. a 3D value on (i, j, k) is simulated using the 1D index $\sigma(i+\alpha_1 * \sigma(\alpha_2+\alpha_3 * j+\alpha_4 * \sigma(\alpha_5+\alpha_6 * k)))$, where the fixed parameters $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6$ are big numbers (generally primes).

To be smooth, the interpolation used should be better than linear. Rather than using cubic interpolation, Perlin change the kind of data to be interpolated: instead of single values, he stores 4 values at the nodes of the 3D grid, that associate a plane to this node (i.e. a quadruplet of random values is used, representing a normal and a height). The resulting value of $s(\vec{x})$ is the trilinear interpolation of the distance of x to the plane defined at each on the nodes on the cell on which x lies.

The turbulent noise function is used as a seed or as a perturbation to give images. E.g. a colormap function can turn the values into colors, threshold functions can generate low or high plates in the curve. The $\vec{x}$ value indexing an image or a simple characteristic curve can be perturbed and turned into $\vec{x}+\alpha * t(\beta * \vec{x})$ were α controls the amplitude of the perturbation and β the frequency of its details[2]. Marble and wood are simulated that way, using a characteristic function that simulates an unperturbed vein for marble (a pick at the location of parallel planes, see figure 1) and for wood (a pick at the location of parallel circles, see figure 2).

2.2 Beyond Basic Graphics Libraries Features

Z-buffer based graphics libraries like OpenGL only implement a restricted set of geometric, photometric and textural representations[NDW93]. E.g. the shapes are only made of triangles, the photometric model is either Gouraud or limited Phong (computed at surface nodes then interpolated on the triangles). The textures are based on the mapping of images using (u, v) texture coordinates given on the surface nodes. These limitations are mainly due to the use of the Z-buffer algorithm, and to the constraints of using hardware (e.g. interpolating and normalizing a normal

[2]A perturbation has 3 components. $t(\vec{x})$ and $s(\vec{x})$ are then vector functions, i.e. they provides 3 uncorrelated noises in each of the 3 dimensions. So α and β can be a number, a diagonal matrix, or a full matrix if anisotropic perturbations are wanted.

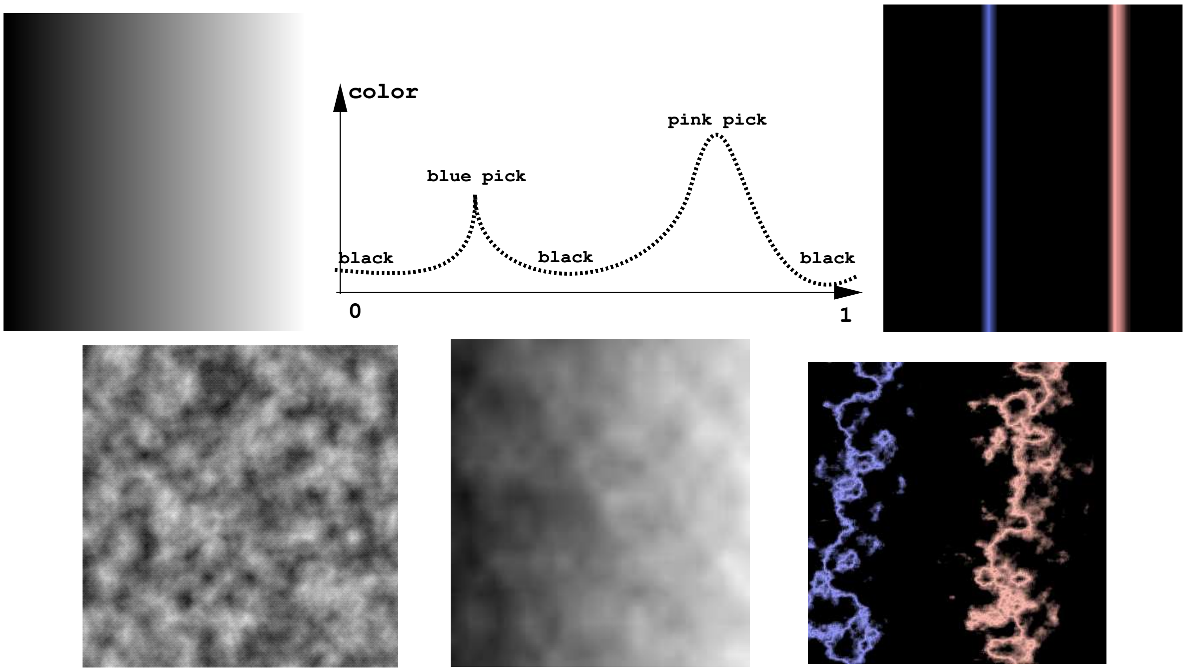

Figure 1: Marble texture: characteristic function $f()$, colormap $C()$, $C(f())$, turbulence $t()$, perturbation of $f()$ by $t()$ (i.e. $f(x+\alpha.t(x))$), result with the colormap (i.e. $C(f(x+\alpha.t(x)))$). NB: these images are produced in real time using our algorithm.

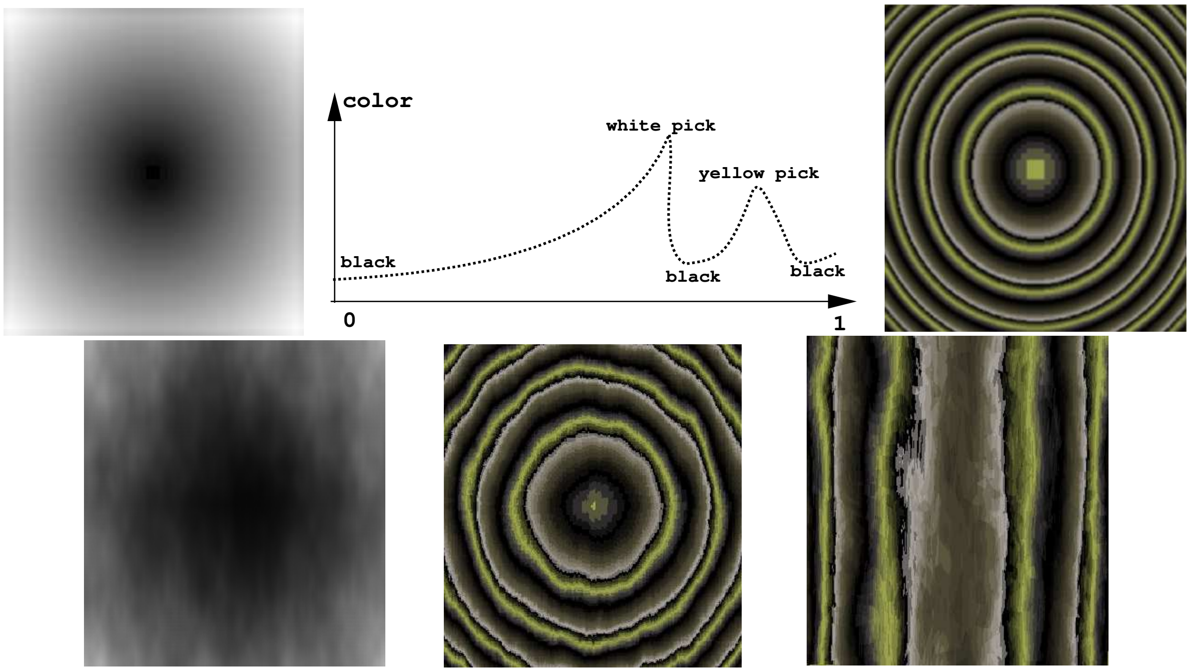

Figure 2: Wood texture: characteristic function $f()$, colormap $C()$, $C(f())$, perturbation of $f()$ by $t()$ (i.e. $f(x+\alpha.t(x))$), result with the colormap (i.e. $C(f(x+\alpha.t(x)))$), same from a side view. NB: this images are produced in real time using our algorithm.

vector along a triangle in order to implement a real Phong shading would need rather more complex electronics). However many open extra features allows for extensions, provided one knows how to translate a problem in terms of the limited grammar provided. E.g. textures coordinates can have from 1 to 4 dimensions, their value is freely determined by the programmer and a 4×4 texture matrix can transform them at rendering time. A 4×4 color matrix can be applied to a resulting color RGBA seen as a 4D vector, multiple color tables can transform the color or alpha value coming from pixels or from a texture, before or after color matrix multiplication, etc. Combinations of transparent layers can also differ for regular compositing by choosing other operators and coefficients than ones of the regular blending equation, and textures are not limited to defining a color: they can be mutiplicative (e.g. to simulate lighting), or contain a Z value.

An important point is to keep in mind that OpenGL ignores the *meaning* of operations and values, it simply *processes* them. Thus, interpolating 1 or 4 texture coordinates along a triangle is an equivalent process, indexing an array with 1 or 4 components is quite similar. It's the user and the programmer who give interpretation to what the texture is attached to, and what the image contents represents. E.g. environment reflections can be obtained by encoding in (u,v) the direction of a reflected ray at a given node, while for the rendering they are (u,v) like any others.

In particular, textures coordinates with more than 2 dimensions can be used in two ways:

- if a 4×4 texture matrix is provided, the coordinates are interpreted like regular coordinates, and transformed using the matrix (e.g. a projection). Then, one may consider the 2 first components of the result to index a texture image, the same way that the 2 first components of the 4D geometric and camera transform are considered to index a pixel on screen.
- by indexing a 3D table with 3 texture coordinates that are a linear transform of the node location, one can display a slice of a volume. The volume is encoded in the 3D texture, and the slice is given by the polygon location in geometric space and texture space. Note that rendering a surface using 3D texture coordinates gives exactly a solid texture as defined previously. Volume rendering can also be obtained by using a sequence of slices and transparent textures.

Using these OpenGL special features or extension, one can implement mirror reflections[NDW93], shadows, Fresnel lighting, bump mapping, volume rendering[WE98], volumetric textures[MN98], and many other effects usually only available in ray-tracing. Many of them are described in the Siggraph Advanced Graphics Courses [CR98] and on the SGI web site [Gra]. In particular, some clues on how to implement basic Perlin's textures are given in [CR98]: The idea is to define a small random 3D texture, and to map it several times while reducing the size by a factor of two, using the `GL_ADD` additive blending with no blending coefficient (i.e. 1 and 1 instead of $alpha$ and $1-alpha$). More functionality is necessary however beyond this basic solid texture in order to get a fully usable Perlin's texture. In particular, one needs to use this basic 'signal' as a perturbation function as explained before, in order to get the veins of marble or wood. This is described in the next section.

3 Perlin's textures using OpenGL

As we have seen in previous work, the more general Perlin texture equation that gives the color (or any other surface feature) at a given 3D location is modeled by:

$$C(f(T_1.(\vec{x}-\vec{x0})+T_2.\vec{t}(T_3.(\vec{x}-\vec{x1})))) \tag{1}$$

with $\vec{t}(\vec{x}) = \frac{\sum_0^n \frac{1}{2^i}.\vec{s}(2^i.\vec{x})}{\sum_0^n \frac{1}{2^i}}$ the turbulence function that produces the perturbation,
$f(\vec{x}) : \mathbb{R}^3 \to \mathbb{R}$ the characteristic function of the material,
$C(x) : \mathbb{R} \to \mathbb{R}^4$ is the colormap that gives an RGBA value,
$\vec{s}(\vec{x})$ is the pseudoperiodic noise function obtained by the interpolation of the random values given at the nodes of a (virtual) 3D grid.
$\vec{x0}$ and $\vec{x1}$ controls the translation of the characteristic pattern and of its perturbation,
T_1, T_2 and T_3 are matrices controlling the orientation and the directional size of the characteristic pattern and of its perturbation.

In order to simplify the computations, we decompose the transformation T_2 into a rotation R_2 and a scaling S_2, and factorize the R_2 rotation. We note α_1, α_2 and α_3 the diagonal coefficients of S_2. This gives the equivalent expression:

$$C(f(R_2.((R_2^{-1}.T_1).(\vec{x}-\vec{x0})+S_2.\vec{t}(T_3.(\vec{x}-\vec{x1}))))) \tag{2}$$

The point is now to translate this equation in terms of OpenGL (or any other rich graphics library) operations. In 3.1 we see how to generate the pseudoperiodic noise $\vec{s}(\vec{x})$, and in 3.2 how to build from it the turbulence $\vec{t}(\vec{x})$. We explain how to obtain a material such as marble or wood from that, by expressing the characteristic functions $f(\vec{x})$ in 3.3, and the color function in 3.4.

3.1 Generating the pseudoperiodic noise $\vec{s}(\vec{x})$

As suggested in the previous work section, we use a 3D texture containing random values to define the 3D grid. The interpolation to get the values at pixels lying between grid nodes is done by OpenGL, by selecting the magnifying filter `GL_LINEAR` . Despite the fact that it is less smooth than cubic interpolation, it gives correct results. In practice we use a $16 \times 16 \times 16$ random 3D texture. As on one hand we only need intensity values, and and the other hand we need uncorrelated values for the 3 dimensions (i.e. $s()$ is a vector), we will encode all along the process these 3 dimensions into the R,G,B channels. Thus, the 3D texture contains random RGB values.

3.2 Generating the turbulence $\vec{t}(\vec{x})$

The various scales of noise are added using multipass rendering: The (u, v, w) texture coordinates values at the nodes of the object to be rendered are initialized with the translated rotated and scaled geometric nodes coordinates $T_3.(\vec{x} - \vec{x1})$. The current color is used to tune the scaling. It is initialized with α_1, α_2 and α_3 stored in R,G and B, divided by 2 to incorporate an approximation of the normalization by $\sum_0^n \frac{1}{2^i}$ (that is 2 for n infinite), then the object is rendered. To process the other passes, we multiply the texture coordinates by 2, we divide the current color by 2, and we render the scene again. For a correct addition to be done, we choose `GL_ADD` and $(1, 1)$ for coefficients in the blend operation. The iteration is repeated as many times as required by the fractal depth n (usually around 4). A very efficient solution to hide the texture repetition consists in rotating the texture at each iteration. As the perturbation size is normally a fraction of the main pattern, the α are much less than 1, so that no overflow will occur.

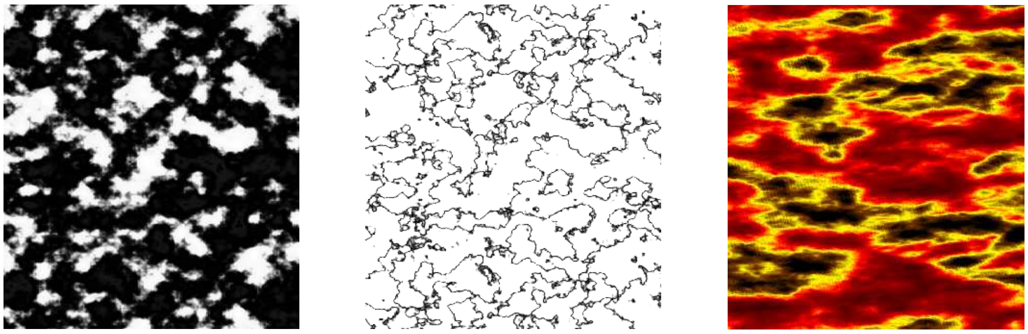

Figure 3: textures with no characteristic function, purely defined by $C(t(x))$.

3.3 Generating characteristic functions $f(\vec{x})$

At this stage, some kinds of textures like grainy rocks or clouds simply need to transform the turbulent noise into color using a colormap (see figure 3). For other materials, the texture is defined by the turbulent perturbation of a characteristic pattern. The perturbation is expressed by $\vec{x}+\vec{t}(\vec{x})$. The characteristic function is $f(\vec{x}) = \vec{x}[0].(1,1,1)$ for marble (see figure 1.1), as ideal features are vertical and parallel. It is $f(\vec{x}) = \sqrt{\vec{x}[0]^2+\vec{x}[2]^2}.(1,1,1)$ for wood (see figure 2.1), as ideal features are vertical concentric cylinders. From that point, we will only deal with these two generic examples.

We already have the perturbing term $\vec{t}()$ computed from the previous section. The evaluation of $f()$ first needs to add to $t()$ the displacement to be perturbed.

Perturbed displacement:
For this we use another 3D texture figuring the identity operator ID_{xyz}, i.e. having u,v,w stored in RGB at each texture pixel location (it is thus a 3D ramp). Then we render the object again, with the additive blend still enabled, after having initialized the (u,v,w) texture coordinates values at nodes with the translated rotated and scaled geometric nodes coordinates $R_2^{-1}.T_1.(\vec{x}-\vec{x0})$ (as required by equation 2. As identity is a separable function, one can also use 3 1D textures figuring ID_x, ID_y and ID_z (in fact it is the same, mapped using only one of u,v or w at a time). This avoids using a 3D texture, but this needs 3 rendering passes. This is useful for low-end graphics cards that do not implement 3D texture in hardware.

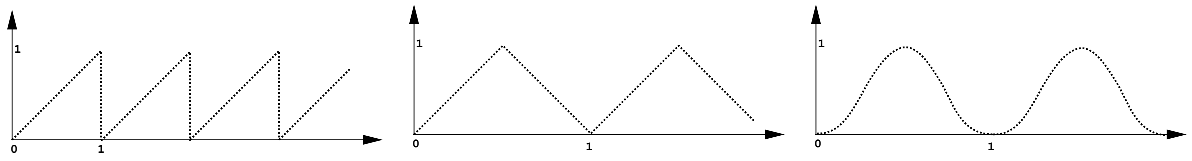

Figure 4: The identity function with modulo, in order to avoid overflow. Two other possibles functions avoiding the modulo discontinuity.

Of course we cannot map the identity in negative values and up to infinity. The function will have a saw tooth periodic shape, that is the same used by OpenGL to repeat texture tiles all along a surface. For some textures, it can be a problem to have the discontinuity due to the modulo. In such cases, we can use a triangular characteristic function instead. A sine can also be used. These 'identity' functions are represented on figure 4.

Overflow can occurs when adding the identity function and the perturbation. Since the last has a limited amplitude A that can be known, we weight the identity function so that it remains below $(1-A)$, i.e. we set the current color to $(1-A).(1,1,1)$.

Marble characteristic function:

As seen above, this function keeps only the $\vec{x}[0]$ component and copies it in the others components (we deal with frame transform in paragraph 3.3). This can be done by applying the transform matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

We use the color matrix, that multiplies RGBA pixels values seen as a vector before storing them in the frame buffer. As this should be done only once the iterative sum it computed, and not on the fly, we have to copy the frame buffer onto itself, in order to activate the color matrix transform. This copy should be limited to the object bounding box area.

To avoid touching the other objects on the screen during this operation on the frame buffer, we use the *stencils*: during the very first object rendering we set the stencil in order to mark pixels covered by the object, then during the copy pixels operations we enable the stencil test. The stencil will be reset to zero during the very last rendering pass.

It should be noted that as long as only the first component $\vec{x}[0]$ is kept, we can simplify the previous computations, replacing vector operations by scalar ones, and directly storing the same value in the 3 components RGB to avoid the final copy we need here. Thus we use a luminance 3D random texture, that OpenGL will understand as R=G=B, and a 1D identity luminance texture containing u at each texture pixel location. After rendering the scene, we directly obtain the result without having to use the color matrix and the frame buffer copy. Avoiding the matrix multiplication and the buffer copy a lot of time can be saved, especially on low-end graphics cards.

Wood characteristic function:

For the wood, we have to compute $\sqrt{\mathbf{x}[0]^2 + \mathbf{x}[2]^2}$. We have first to build the squares or the channels R and B. This can be done by copying the screen onto itself while using the blending in a very special way: we as a blend coefficient the image itself, which produces the squares (triangular identity function is used, to get the symmetry around (0,0,0)). Then we use the color matrix to sum the 2 squares and copy the result in all the components, with the matrix

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

This implies that the frame buffer must be copied again onto itself, as explained above (the blend operation being implemented in the pipeline after the color matrix transform, we alas cannot achieve these 2 operations in a single pass). The square root may be computed using a color map that would transform x into $\sqrt{x}$. This can be done in the same pass as the color matrix transform. As we already use a color map for the color, we will rather compile these two maps into one before rendering.

Including the rotation R_2

As the rotation R_2 has to be done before the evaluation of $f()$, as specified by equation 2, actually each of the noise components will be used. Rotation can simply be included in the color matrix, by multiplying this matrix by the rotation ([WE98] does a large amount of geometric transforms using the color matrix, including rotations, and even computes the Lambert shading with it).

To be noted that for most textures, the transform R_2 is the identity matrix (T_2 is only a diagonal scaling matrix): it is rare to want a preferred direction of distortion that is different from the preferred direction of texture 'grain' (i.e. frequency) encoded in T_3.

3.4 Generating of a material with $C(x)$ and $f(\vec{x})$

As seen in the previous work, the color map not only gives colors to the texture, but it really defines its main features, by selecting picks and plates, i.e. particular ranges of values. OpenGL provides for color maps, that can even be used to increase a texture resolution (the interpolation generates values between the texture pixels, which are individually considered in the colormap). We let the user define some key RGBA values that we interpolate to produce the map. In the wood case, we store a key designed for a value c at the location c^2 in order to take into account the $\sqrt{}$ transform. At rendering time the colormap is applied with a copy of the screen buffer onto itself, which can be done on the same pass as the previous color matrix transform.

4 Summary

Here is a summary of the algorithm, for the more complicated case that is the wood texture.

```
# --- Perlin noise pass
current color = (alpha1/2,alpha2/2,alpha3/2)
text coord at each node = T3.(x-x1)
set blend = ADD, coefs = 1,1
enable stencil set
iterate 1 to 4 times:
   render
   multiply text coord by 2
   divide current color by 2
# --- identity to be perturbed added
A = MAX(alpha1,alpha2,alpha3)
current color = (1-A,1-A,1-A)
text coord at each node = inv(R2).T1*(x-x0)
render
# --- prepare the squares
set blend = ADD, coefs = SOURCE,0
enable stencil test
copy frame buffer onto itself
# --- process c(f())
set color matrix to F_wood.R2
set the colormap
enable stencil reset
copy frame buffer onto itself
```

5 Results

As seen in the previous section, the whole process requires 2 to 5 rendering passes, depending on the noise frequency range required, and also 2 block copies in the frame buffer.

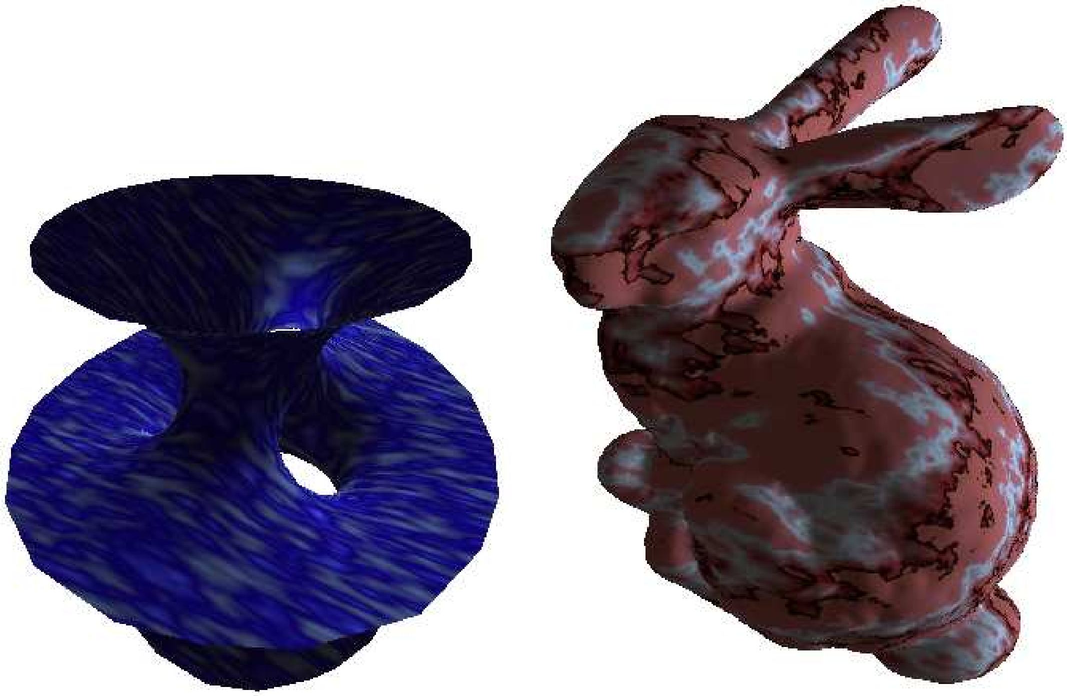

Figure 5: Left: minimal surface, 919 faces, about 35 frames per second. Right: the 'Bunny' big mesh, 70,000 faces, about 2 to 3 frames per second.

For the minimal surface in figure 5.1 containing about 900 faces, we get 30 to 40 images per second on Onyx2 Reality Engine. For the well-know decimation test object 'Bunny' on figure 5.2 containing about 70,000 faces, the rendering is no longer real time, but still interactive with a few images per second. A remind is that simplest scenes are ray-traced in at least ten minutes.

We have also test the program on a low-end architecture: The O2, that does not contain any hardware for 3D textures, color maps and color matrix. We replace the 3D texture by a 2D texture for the tests, which give sometime sufficient effects (otherwise it kills the frame rate !). The geometry on figure 5.1 is then obtained at 10 images per second. On O2 the rabbit already needs 1 second to render with ordinary rendering. With Perlin noise, it needs several seconds.

6 Conclusion and Future Work

In this paper, we have presented a complete solution to the synthesis of Perlin noise in real time on standard OpenGL meshes, using advanced OpenGL features. Although some of these features are not yet implemented in hardware on low-end graphics cards, they are simple and generic enough that they have a chance to appear soon on a wider market. This allows us to add a great amount of realism in real-time for interactive applications, and to greatly speed-up the procedural texture pass for realistic rendering. A library extending OpenGL has been developed from this work, that will soon be made publicly available.

As future work, we consider the translation in hardware of other procedural textures, such as Worley's textures[Wor96] that allows to produce nice cellular shapes such as scales or rocks, also limited for the moment to the context of non real-time quality rendering such as ray-tracing (being pixel-based). The principle of this technique consists in choosing random points on surface or in volume, and to consider nearest neighbor areas (i.e. Voronoï regions), to be combined with lower order nearest neighbors. The per-polygon translation of this, in order to be OpenGL compatible, would consist in using a texture containing a distance map represented by a concentric ramp, centered on each random point, and to use the `MINMAX` blend extension to keep only the min distance value.

References

[Arv90] J. Arvo. *Graphics Gems II*, chapter ,.10, page 396. Academic Press, 1990.

[CR98] Siggraph Course Notes CD-ROM. *Advanced Graphics Programming Techniques Using OpenGL*. Addison-Wesley, 1998. http://www.sgi.com/software/opengl/advanced98/notes/notes.html.

[EMP+98] Ebert, Musgrave, Peachey, Perlin, and Worley. *Texturing and Modeling, a Procedural Approach*, chapter 2, page 66. AP Professional, 1998.

[Gra] Silicon Graphics. Performer white papers. http://www.sgi.com/software/performer/whitepapers.html.

[KK89] 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.

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

[NDW93] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993.

[Per85] Ken Perlin. An image synthesizer. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 287–296, July 1985.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 169–178. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

[Wor96] Steven P. Worley. A cellular texturing basis function. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 291–294. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399