INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

All-Purpose Texture Sprites

Sylvain Lefebvre
EVASION/GRAVIR

Samuel Hornus
ARTIS/GRAVIR

Fabrice Neyret
EVASION/GRAVIR

http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/texsprites

N° 5209

May 2004

Thème COG

Rapport de recherche

N 0249-6399 ISRN INRIA/RR--5209--FR+ENG

All-Purpose Texture Sprites

Sylvain Lefebvre — EVASION/GRAVIR
Samuel Hornus — ARTIS/GRAVIR
Fabrice Neyret — EVASION/GRAVIR

http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/texsprites

Thème COG — Systèmes cognitifs
Projets EVASION et ARTIS

Rapport de recherche n° 5209 — May 2004 —20 pages

Abstract: We propose a representation for efficiently and conveniently storing texture patches on surfaces without parameterization. The main purpose is to texture surfaces at very high resolution while using very little memory: *patterns* are stored once while instance (*i.e., sprites*) attributes (pattern number, size, orientation) are stored in an octree-like structure (requiring no surface parameterization). Our representation correctly handles filtering while most other methods suffer from filtering artifacts at patch boundaries.

Implemented as texture sprites, the texture patches of a composite texture can be updated dynamically. This provides natural support for interactive editing, and also enables various kinds of animated textures, from wavy stationary features to crawling spots. We extend this basic scheme with two examples which would be uneasy to achieve with other methods: complex blending modes between the texture patches, and rigid scales on a deforming surface.

Since our representation is particularly well suited for interactive applications and texture authoring applications, we focus in the paper on its GPU implementation, while preserving high-quality rendering.

Key-words: texture sprites, texture storage, animated textures, hierarchical structures, GPU

Textures composites dynamiques

Résumé : Nous proposons une représentation pour stocker efficacement et facilement des morceaux de texture sur des surfaces non paramétrées. L'objectif principal est de texturer des surfaces avec une très haute résolution, tout en utilisant peu de mémoire: les morceaux de texture sont stockés une seule fois alors que les attributs (position, taille, orientation) des instances (appelées "sprites") sont stockés dans une structure hiérarchique similaire à un octree (ce qui ne nécessite pas de paramétrisation). Notre représentation gère correctement le filtrage de la texture composite ainsi obtenue alors que la plupart des autres méthodes souffrent de défauts de filtrage à la bordure des morceaux de texture.

Implémentés sous la forme d'instances, les morceaux d'une texture composite peuvent être mis à jour dynamiquement (position, taille, ...). Ceci permet l'édition interactive de la texture et autorise la création de textures animées, allant d'éléments stationnaires animés à des motifs se déplaçant sur la surface. Nous étendons ce schéma de base vers deux exemples difficiles à réaliser avec d'autres méthodes: le mélange complexe de morceaux de texture et des écailles rigides sur une surface déformable animée.

Étant donné que notre représentation est particulièrement adaptée aux applications interactives et d'édition de textures, nous nous concentrons dans l'article sur son implémentation pour les processeurs graphiques récents, tout en préservant un rendu de haute qualité.

Mots-clés : texture composites, stockage de textures, textures animées, structures hiérarchiques, GPU

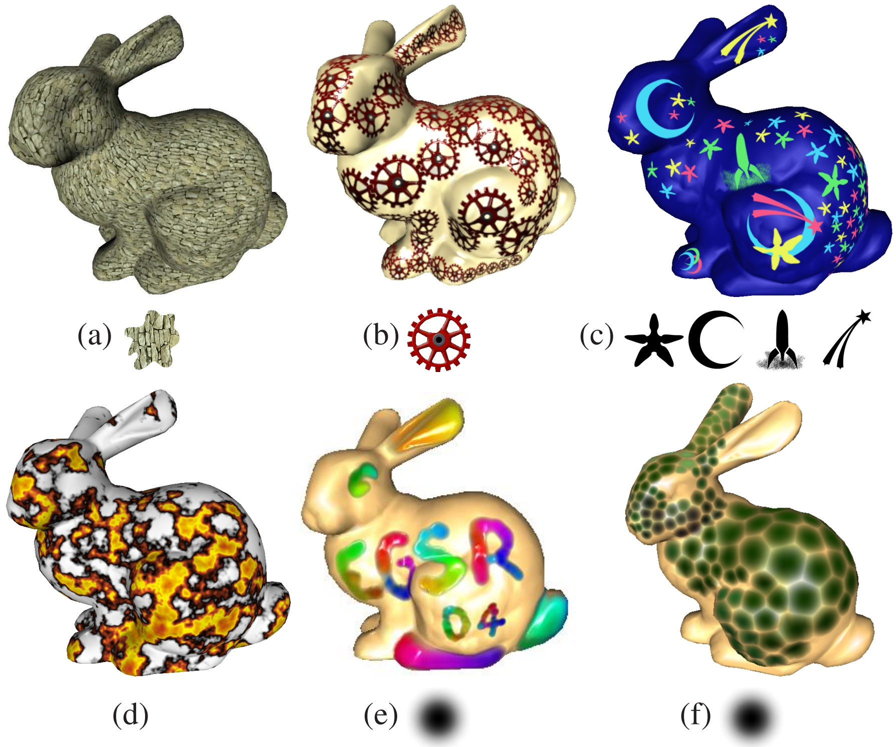

Figure 1: *From left to right*: (a) Lapped textures. (b-c) Animated sprites. (d) Octree texture. (e) Blobby painting. (f) Voronoi blending of sprites. The bottom line shows the texture *patterns* used (the only user defined textures stored in video memory). All these bunnies are rendered in one pass, in real-time or at interactive frame rates, using our *composite texture* representation. The *texture sprites* can be edited interactively. The original low-poly mesh is unmodified.

1 Introduction

Textures are crucial for the realism of scenes; they let us add details without increasing the geometric complexity in real-time applications. Games and special effects now rely on multiple texture layers on objects.

However, creating and mapping textures is not trivial and has motivated numerous articles and systems. The surface parameterization used for the mapping has long been an issue for the users since it deforms the texture content. The problem has been reduced by *texture atlases* [MYV93] which attenuate the distortions by cutting the texture in small unstretched pieces. Interactive 3D paint systems [HH90, Bod, Dee, Tex] are also able to cancel the distortion by taking mapping deformation into account during texture painting. However, an underlying parameterization is still used and influences the needed texture resolution: a dilated area requires to increase the overall resolution while a compressed area does not make use of the full existing texture resolution.

Apart from painting tools, the current trend in texture synthesis is to re-synthesize large textures from image samples. Most of them [WL00, EF01] explicitly store the resulting texture in a large image, which wastes a lot of memory since the patterns are actually repeated in the image. Some methods proceed directly on an object surface [PFH00, Tur01, WL01, SCA02].

Being an alternative to geometric details, textures are also a solution to encode scattered objects like footprints, bullet impacts [Mil00], or drops [NHS02]. These are likely to appear dynamically during a video game. Doing this by updating a texture layer would require to texture the whole surface even if the event is local. Moreover, animating these objects (e.g. rolling drops) can lead to tricky texture updating. The common solution is to use *decals* instead, i.e., to put the texture on extra small textured transparent quads. However, such marks do not stick correctly to curved surfaces and intersection between decals yields various artifacts. This gets worse if the underlying surface is animated since all the decals must be updated at each time step.

We describe a new representation for textures that addresses all theses issues, and provides a general and convenient scheme for pattern-based texturing. It can be implemented on recent GPUs, and hence is practical for real-time applications. We demonstrate that it deals correctly with filtering while most other methods suffer from filtering artifacts at patch boundaries. It does not require that the mesh conforms to any constraint contrary to most other methods. It does not require to introduce new geometry.

The principle is to store only the patterns positioning attributes and to make the renderer determine at rendering time which color to use at each pixel. We draw on [DGPR02, BD02] for the parameterization-free painting of attributes and on [LN03] for the pattern positioning. We describe a complete GPU implementation with details to solve precision and filtering issues. Since our objects live in texture space and can be updated dynamically we will refer to them as *texture sprites* after [NHS02].

The contributions of this paper are:

- A new texture representation allowing efficient storage and updating of composite textures in interactive and real-time applications.
- An efficient, unified solution to numerous problems such as texture memory saving, decals management, animated textures and filtering issues at tile or atlas boundaries.
- As a case study, a solution dedicated to storing lapped-like textures with high resolution and little memory.
- A complete GPU-implementation of our representation, based on a hierarchical 3D texture.

2 Previous work and texturing issues

The easiest way to texture an object is to map an image on its surface. Unfortunately this is more complicated than it first appears:

- If objects can be seen from close viewpoints, the texture resolution needs to be high (possibly higher than the screen resolution).
- In the common case of non-developable surfaces, the mapping is distorted. The image can be painted manually or automatically in order to cancel the distortion. However this requires an increase in the overall texture resolution in order to still have enough resolution in dilated areas. Classical solutions to texture mapping and texture storage issues are discussed in section 2.1.
- If objects can be seen from far viewpoints, the texture needs to be properly filtered in order to avoid visual aliasing. The filtering issues are discussed in section 2.2.

2.1 Applying textures

Classical solutions and their issues
Concerning the distortion problem, note that optimizing the (u,v) texture parameterization can improve it but not solve it: a distortion always remains. A solution is to cut the mapping to allow each part to release the stress more easily. This corresponds to using tiles (separate regular pieces of texture) or a texture atlas [MYV93] (an image in which the texture has cracks and possibly isolated parts, the pieces being stuck together by the mapping – see figure 2).

A solution to save memory while keeping high resolution is to rely on repeated patterns using tiles (see figure 2). Games usually use quad tiles for the terrain (it is possible to avoid regularity with a small set of tiles as seen in [CSHD03, LN03]). Triangular texture tiles have been used in [NC99] for other kind of objects. Note that a texture atlas also allows to reuse a given area at several places and to adapt the resolution of each area, which is commonly done in games. Numerous tools now help to paint textures on 3D models [Bod, Dee, Tex]. They generally start by automatically cutting and optimizing the parameterization. However game programmers often prefer to set the (u,v) values and to draw the atlas manually.

Figure 2: *left*: A *tiling* allows to save memory by instancing a small set of patterns. *right*: A texture *atlas* releases distortions by allowing mapping discontinuities.

We will show in section 2.2 that the mapping discontinuity in tiles and atlases introduces filtering artifacts. Moreover, note that these cuts can only be done along the mesh edges. This either constrains the meshes (vertices location) or introduces distortions in the texture.

Storing composite textures
Numerous algorithms have been recently proposed to automatically produce large textures from an image sample [WL00, EF01, KSE*03]. Some of them proceed directly on the surface without requiring any parameterization [PFH00, Tur01, WL01, SCA02, DMLG02]. Since the same image elements are reused many times, it is a huge waste of memory space to reconstruct a large image

to be mapped. Authors proposing better solutions for the storage either generate a soup of small textured polygons [DMLG02] or redraw several times the faces on which several image elements are mapped [PFH00]. As explained in this last paper, instancing instead of duplicating texture elements allows to store a far better resolution with the same (or less) amount of memory. However drawing several superimposed faces increases the rendering cost and yields various blending issues and Z-buffer conflicts. This is even worse with animated geometry. The dynamic creation and update of these small geometry patches may also be difficult.

2.2 Filtering issues

The purpose of this section is to shed light on classical filtering issues that are often neglected. Not addressing these issues can result in various rendering artifacts.

It is important to realize that GPU rendered scenes are anti-aliased mostly thanks to textures and despite aliasing in the rasterized geometry (assuming no oversampling is done): Only the center of pixels are sampled, but the spatial coherency of the texture allows the GPU to guess the correct average color within a pixel. However this works only under the assumption that the texture mapping function is continuous along the surface.

Discontinuities

Each time a mapping discontinuity is defined in the mesh, the assumptions above are broken, and the filtering is no longer correct, both for linear interpolation and MIP-mapping. This issue typically occurs when two triangles sharing an edge have different textures or use different regions of the same texture. The latter case occurs in *textures atlases* (see Figure 3).

These artifacts are often hidden by artists by modifying the background in the texture image or by adding thick borders (i.e. propagating) around texture features. This does not solve the problem for two reasons. First, there always exist a MIP-mapping level showing the artifact, i.e. when the footprint of the pixel becomes larger than the added border. Second, adding this border results in a larger texture which wastes memory.

Note that oversampling only decreases the artifacts since non correct texture areas are still averaged during texture lookups.

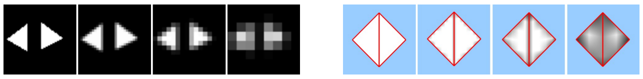

Figure 3: *Left*: four MIP-map levels of a texture atlas containing two regions. *Right*: The two regions are mapped on two adjacent triangles (*red*). One can see that texture background color bleeds at the edges.

Limits of texture filtering: geometric aliasing

Only the center of pixels is sampled (assuming no oversampling is done). So, when faces get smaller than pixels – which occurs easily now that the graphics hardware power enable the use of dense meshes – several faces of a mesh can project in a given pixel but only one face is actually sampled. Things can do well only if the texture of the sampled face is representative of what occurs in all the pixel, i.e. the faces share the same texture mapping or rely on different textures sharing the same color statistics. In all the other cases, bleeding of wrong colors and possibly aliasing will occur. Here again oversampling do not solve the issue: aliasing will still occur but for smaller faces. Creating small patches of geometry for the patterns of a composite texture is therefore likely to produce aliasing. This can occur even for close viewpoints if the texture patterns size is small compared to the object size.

The new powerful features of GPUs are adding new filtering issues since numerous non-linear or discontinuous effects can now occur within a face: indirections can be used to cover two nearby areas with different texture regions. Pixel shaders can modify or procedurally determine the pixel color. In both cases, there is no longer any embedded mechanism that is able to estimate a correctly filtered value: it is the programmer's responsibility to handle the filtering. Every pixel shader programmer has to be aware of that! Even so, detecting and fixing the problems are not easy [KE02]. Discussion and partial solutions about these issues can be found in [LN03].

3 Our representation

Our goal is to obtain a convenient system to represent and render composite textures.

To avoid the artifacts of the previous models, we want to have low texture memory cost, no distortion due to global parameterization, no added geometry other than the initial mesh involved, and a correct filtering (interpolation and MIP-mapping) in all cases.

Moreover, we would like to be able to interactively edit the composite texture, control the blending of overlapping sprites, and allow animated textures.

Of course, we still want the resulting model to be rendered in real-time, and support animated meshes.

We draw on the dynamic pattern positioning scheme proposed in [LN03] since it fulfills the low memory cost requirement without introducing new geometry. The idea in [LN03] is to rely on a list of patterns and on indirection textures containing the pattern number and positioning parameters to describe the texture.

However this method works in 2D and thus relies on a global parameterization of the surface. Moreover overlapping sprites are not handled, and could not be trivially added. Therefore, this method cannot represent composite textures and is no more adapted to curved surfaces than classical mapping.

We have seen in section 2 that the classical solutions to mapping distortion introduce mapping discontinuities which yield filtering issues. We rely on a totally different mapping approach: surface parameterizations suffer from distortions and/or discontinuities but volumetric parameterizations do not. Up to recently volumetric parameterizations were used only for solid texturing [Per85, Pea85, Wor96] and volume rendering. To avoid the problems of parametric distortion, we follow the idea of *octree textures* [DGPR02, BD02], which relies on an octree to store the surface color (only nodes intersecting the surface are created).

Our key idea is to store the sprite positioning parameters in a 3D hierarchical structure. This avoids the need for global parameterization in the spirit of [DGPR02, BD02], while providing the advantages of the texture representation of [LN03]. Several difficulties have to be solved: defining how to map the sprites on an object surface, how to store overlapping sprites while allowing dynamic updating, and how to render them so to avoid filtering artifacts.

Our representation consists of:

- A list of patterns (*i.e.,* images).
- A 3D hierarchical grid. Instead of octrees we will rely on N^3-trees. Our nodes are meant to contain texture sprites and not pixels, so the average depth of the tree will be low. **(section 4.1)**
- A node structure able to contain several sprites (with positioning information plus various attributes for each sprite). **(section 5.1)**
- Customizable projector functions able to interpret the positioning information in order to associate a texture sprite (u,v) location to a 3D point lying on the surface. **(section 6)**
- Customizable blending operators able to treat overlapping sprites according to the user requirements. **(section 7)**

Our representation is interesting for quality software renderers, to reduce memory requirements of composite textures and to avoid rendering artifacts. Since it is also amenable to GPU implementation despite the more constrained context, in the following we focus on the description of a GPU implementation (it is easy to adapt it to a software framework). Sections 4 and 5.1 describe how to manage the 3D-tree on the GPU so that a given pixel fragment can access the texture sprites information. We provide details on how to reduce memory storage, avoid precision issues and rendering artifacts in sections 4.3 and 8. The construction of the 3D-tree from the sprites is described in section 5.2. Note that this GPU implementation offers the same high quality (i.e. concerning filtering) than a software implementation would.

4 Implementing a N^3-tree on GPU

Hierarchical grids such as octrees are often used to compactly store sparse data (e.g. images or volumes). They are easy to implement using arrays. The texture indirection ability of modern GPUs is quite similar to the notion of an array, and the programmability of these GPUs enables the programming of the structure access. However there are numerous limitations (GPUs are not general purpose CPUs), it is crucial to think about the consequences of each choice in terms of performance, and numerous precisions issues complicate the task (low dynamics of values, indices are floating points).

[KE02] introduced the first attempt (using a first generation programmable GPU) to skip the empty space in a target texture by packing blocks of data so as to only store a small texture on the graphics board. They used an indirection method based on a regular grid: To encode a virtual texture T containing sparse data the space is first subdivided into n tiles. Then the non-empty tiles are packed in a texture P and an indirection map I of resolution n is used to recover the pixel color of any given location: $T(x) = P(I(x) + \frac{frac(x \cdot n)}{n})$. ($\lfloor x \rfloor$ and $frac(x)$ denote the integer and the fractional parts of x). I can also encode an offset rather than an indirection, in which case $T(x) = P(I(x) + x)$. [LN03] shows how a similar indirection method can be used to arbitrarily position and instantiate sprites in texture space: the texture storage is considered as a list of patterns and the indirection grid tells, for each cell mapped on the surface, which pattern to use and with which parameters to tune it. Only one pattern can be stored in each cell.

We extend these two ideas to create a *hierarchical* 3D grid (*i.e.,* a tree) storing sprite positioning information.

4.1 Our N^3-tree model

Tree structure

Figure 4 shows the grid structure: each node represents a *indirection grid* consisting of a small number of cells; the cell content(whose type is determined by a flag stored in the A-channel) is either (a) *data* if it is a leaf ($A = 1$), or (b) a *pointer* to another node ($A = \frac{1}{2}$). A third possibility ($A = 0$) indicates that the cell is empty.

To optimize for the limited range of texture indices, we store our structure in a 3D texture called the *indirection pool*, in which both *data* values and *pointer* values are stored as *RGB*-triples.

Note that this vectorial organization does not add any cost since the GPU hardware proceeds to vectorial operations on 4D vectors. In the following, most scalars have thus to be thought as vectors and conversely (operations are done component by component).

Notations

Let $m \times m \times m$ be the resolution of the indirection grid corresponding to each tree node. An octree corresponds to $m = 2$ (in our implementation we use $m = 4$). The resolution of the virtual 3D grid at level l is $m^l \times m^l \times m^l$. At level l a point M in space ($M \in [0,1[^3$) lies in a cell located at $M_{l,i} = \frac{\lfloor M \cdot m^l \rfloor}{m^l}$ and has a local coordinate $M_{l,f} = \frac{frac(M \cdot m^l)}{m^l}$ within this cell. We have $M = M_{l,i} + M_{l,f}$.

Nodes are packed in the indirection pool. Let n_u, n_v and n_w be the number of allocated nodes in the u, v and w directions. We define $n = (n_u, n_v, n_w)$ as the size of the indirection pool using the vectorial notation. The resolution of the texture hosting the indirection pool is therefore $m\, n_u \times m\, n_v \times m\, n_w$. A texture value in the indirection pool is indexed by a real valued $P \in [0,1[^3$, as follows:

P can be decomposed into $P_i + P_f$ with $P_i = \frac{\lfloor P \cdot n \rfloor}{n}$ and $P_f = \frac{frac(P \cdot n)}{n}$. P_i identifies the node which is stored in the area $[P_i, P_i + \frac{1}{n}[$ of the indirection pool . P_i is called the *node coordinates*. P_f is the local coordinate of P within the node indirection grid.

Note that packing the cells does not change the local coordinates: at level l, if the value corresponding to the point M is stored at the coordinate P_l in the indirection pool then $M_{l,f}\, m^l = P_{l,f}\, n$. It comes:

$$P_{l,f} = \frac{frac(M \cdot m^l)}{n} \tag{1}$$

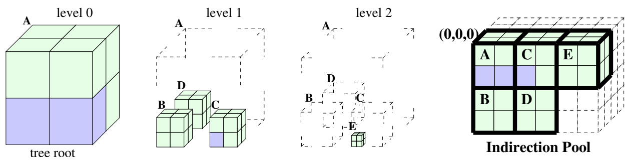

Figure 4: A example tree. The five nodes A,B,C,D and E are packed in the indirection pool (*right*). In this example each node is an indirection grid of size 2^3 ($m = 2$). The cells of the indirection grids contain either data (*light green cells*) or an indirection (i.e. a pointer) to a child node (*dark blue cells*).

4.2 Retrieving values for a given location: tree lookup

The virtual 3D texture in which the data is stored is aligned with the bounding box of the object to be textured. We want to get the data corresponding to a given location M in space (more precisely, on the object surface).

We start from level 0 in the tree. The indirection grid associated with the tree root is always stored at coordinates $(0,0,0)$ in the indirection pool. At level l let $P_{l,i}$ be the coordinate of the current node in the indirection pool. The *RGBA* value corresponding to M is found in the current node indirection grid at the local coordinate $P_{l,f}$, obtained by eqn (1). This eventually corresponds to the coordinate $P_l = P_{l,i} + P_{l,f}$ in the indirection pool.

The value in A tells us whether we are in a leaf, an empty node or an internal node. If we are in a leaf the data *RGB* is returned. If we are in an empty node the fragment is discarded (i.e. the shader aborts). Otherwise *RGB* contains a pointer to a child node and we decode the next node coordinate $P_{l,i}$ from the *RGB* value. Then we continue to the next level $(l+1)$. Our algorithm is described in figure 5.

```
float4 rgba = < zeroes >;
for (float i=0; i<TREE_MAX_DEPTH; i++){
  float3 P = (frac(M)+rgba.xyz*255.0)*inv_n; // child location
  rgba = (float4)tex3D(PoolTex,P); // access ind'n pool
  if (rgba.w > 0.9) break; // type of node = leaf
  if (rgba.w < 0.1) discard; // type of node = empty
  M=M*m; // go to next level
 }
return rgba;
```

Figure 5: The pseudo-algorithm of our tree lookup. M is provided by the CPU via the texture coordinates. inv_n is set to `inv_n=` $\frac{1}{n}$. As the `break` statement is not yet available in Cg our real pixel shader is a bit more complicated.

4.3 Implementation details

Addressing precision and storage issues

To avoid precision issues and minimize storage requirements we store $nP_{l,i}$ instead of $P_{l,i}$ in the indirection cells: since it is an integer we can store it as such, which has two advantages. First, it let's us optimize the encoding: to encode n indices we can choose accordingly the number of bits to be stored in *RGB* while floats would require 4 bytes for each. Second, the values of the pointers do not depend of the size of the indirection pool. The size of the indirection pool can therefore be changed without recomputing the pointers.

Precision limitations
Due to the limited precision of 32 bit floating point values there is a limit on the depth, resolution and number of nodes of our hierarchical grid. To avoid precision issues we use power of two values for m and n. Writing $m = 2^{e_m}$ $n = (2^{e_n}, 2^{e_n}, 2^{e_n})$ and using standard floating point values with a mantissa of 23 bits, the deepest reachable depth is $d_{max} = \lfloor \frac{23-e_n}{e_m} \rfloor$. (see figure 6).

m	n	d_{max}	max. reachable resolution	full indirection pool size
2	16^3	19	$(2^{19})^3$	128 Kb
4	16^3	9	$(2^{18})^3$	1 Mb
8	16^3	6	$(2^{18})^3$	8 Mb
2	32^3	18	$(2^{18})^3$	1 Mb
4	32^3	9	$(2^{18})^3$	8 Mb
2	128^3	16	65536^3	64 Mb
4	128^3	8	65536^3	512 Mb

Figure 6: Typical values of m, n and corresponding limits. The maximum resolution of the resulting 3D texture is $m^{d_{max}} \times m^{d_{max}} \times m^{d_{max}}$. The maximal number of nodes the indirection pool can store is 2^{3e_n}.

Increasing the size m of the tree nodes permits to store more information at a given tree depth. Usually for a same set of sprites this reduces the tree depth resulting in better performances (less texture lookups are required to go down the tree). This also limits the maximum reachable depth d_{max} because of precision limitations. An application limited by storage would rather choose small m value to pack texture data as tight as possible. However for applications where rendering performance is crucial, a greater value of m would be preferable to limit per-fragment computations while still offering high resolution in deepest tree levels. Increasing the maximum number of nodes n that can be stored in the indirection pool permits to create larger structures. An increase of n also reduces d_{max}, but by a smaller amount.

5 Managing the texture sprites

Each sprite is associated with various parameters (including positioning parameters described in section 6) and a bounding volume V defining the spatial limit of the sprite influence on the composite texture. This bounding volume is used to determine which cells are covered by the sprite and to detect whether two sprites overlap.

5.1 Sprite storage

Storing sprite parameters

The parameters associated with a sprite (i.e., with an instance of a pattern) must be stored in the tree leaves. These parameters are a set of p floating point numbers. Since a given sprite is likely to cover several leaves it is not a good idea to store directly this set in each leaf: this would waste memory and complicate the update of sprites. Instead we introduce an extra indirection: The *sprite parameter table* is a texture containing all the parameters for all sprites. Each sprite corresponds to an index into this texture; the texel encodes a parameter set. (Current texture formats allow contents of up to eight 16-bit floats per texel. If more parameters are required it is easy to add follow-up tables to be read in parallel.) We store the index of the sprite parameters in the tree leaf data field: the *RGB* value encodes a pointer to the sprite parameter table just as it encodes pointers to child nodes.

The sprite parameter table has to be allocated in video memory with a chosen size based on the expected number of sprites M. If more are added, the table must be reallocated and old values recopied.

Dealing with overlapping sprites

Not only can each sprite cover several cells but several sprites are also likely to share the same cell. Therefore we must be able to store a vector of sprite parameters in each leaf.

We address this by introducing a new indirection map: The *leaf vectors table*. This 3D texture implements a table of vectors storing the sprite parameters associated with a leaf. Two dimensions are used to access the vector corresponding to a given leaf. The third dimension is used to index the vector of sprite parameters. Each voxel of this 3D texture therefore encodes a pointer to the sprite parameter table. We use a special value to mark unused entries in the vectors. The maximum size O_{max} of a vector controls the maximum number of sprites allowed per cell.

The tree also has to be modified: Instead of directly storing in the leaves the index of one sprite, we now store an index to the leaf vector. The overall structure is illustrated in figure 7.

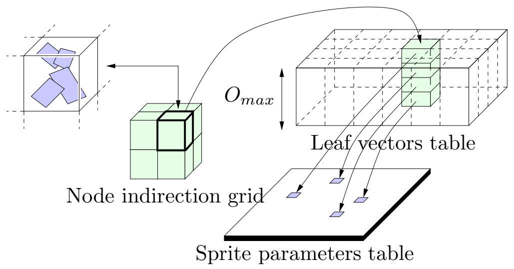

Figure 7: Each indirection cell stores the index of the corresponding leaf vector. Each cell of the leaf vector stores the indices of the sprite parameters.

5.2 Adding a sprite to the tree

The influence of a sprite is limited to its bounding volume. The sprite must therefore be present in all the leaves colliding with this volume. Our insertion algorithm (run by the CPU) is as follows:

```
addSprite(node n, sprite s) :
 if (n is a leaf)
   if (number of sprites in n < Omax) insert s in n
   else
    if (s overlaps with all the sprites in n)
     error(Omax too small !)
    else {
     if (max depth reached) error(Max depth reached !)
     split n
     addSprite(n,s) }
 else
    forall child c of n colliding with s
    addSprite(c,s)
```

The leaves are not split until the limit of O_{max} sprites is reached. Leaf splitting occurs only in full leaves. In our implementation we used $O_{max} = 8$. Generally when a leaf is filled with O_{max} sprites, the maximum number of sprites really overlapping at a given point is $C_{max} < O_{max}$. When inserting a new sprite it is then possible to recursively subdivide the node so than C_{max} is kept smaller than O_{max} in the new leaves (see figure 8). This may occasionally generate a lot of nodes locally if the sprite regions are hard to 'separate'. However, this scheme tends to produce a tree of small depth since most leaves will contain several packed sprites.

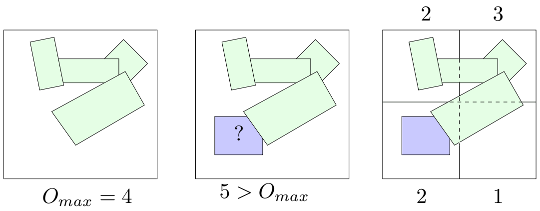

Figure 8: A new sprite (*dark*) is inserted in a full leaf. As the sprite does not overlap all the sprites (*light*), there is a level of subdivision at which the new sprite can be inserted. Subdividing let us keep the sprite count less or equal to O_{max}.

Insertion failures

O_{max} should be chosen greater than the maximum number of sprites allowed to contribute to a given pixel. A painting application might choose to discard the weaker sprite contributions in overcrowded cells to keep this number reasonable.

The maximum depth level is reached only when more than O_{max} sprites are crowded in the same very small region (i.e., they cannot be separated by recursive splitting). Indeed, this is the case where the discard solution mentioned above applies.

Ordering of sprites
The ordering of overlapping sprites might be meaningful in some cases (e.g. snake scales). We ensure that the shader will visit them in the right order, by ordering the sprites in the leaf vectors table.

6 Positioning the sprites on the surface

At this stage we can know which sprite is at each location on the object surface, but no more. In this section we describe how to encode and recover their precise location and orientation. The pattern associated with the sprite must be mapped to the surface, i.e., we need to define a *projection* from texture space to surface space. In practice the opposite is done: the rasterization produces a 3D point and we must project it to texture space.

We associate a frame to each sprite and use a parallel projection to the surface (defining the projection parameters is the responsibility of a mapping tools). So the positioning parameters will be:

- a center point C (sprite handle).
- two vectors L, H defining the plane, scale and orientation of the pattern.
- a normal vector N defining the direction of projection.

Let M denote the matrix (L, H, N). Once the sprite parameters are fetched for a given point P, the pixel shader computes $U = M^{-1} \cdot (P - C)$ with $U = (u, v, w)$ to get the texture coordinates within the pattern. w is useful for volumetric or solid textures, but also in the 2D case: when two faces of the mesh are very close to each other (e.g., a folded surface) we need to distinguish which texture applies to which fold. Instead of forcing the tree cell to subdivide we can simply define a narrow region where the projection applies by tuning the scale of N.

As discussed in [DGPR02, BD02] thin objects or two-sided sheets lead to ambiguous color assignment. In our case we can proceed to a "back-face culling" relying on the direction of projection to distinguish the side on which a given sprite should apply.

The sprite projection on the surface is equivalent to defining a local linear parameterization (note that defining a set of independent local parameterizations is totally different than defining a global parameterization, i.e., a mapping). If the distortion is too high for a given sprite (e.g. when a geometric feature is smaller than the sprite), it is possible to split it in multiple subsprites to minimize distortion. Each subsprite displays only a subregion of the initial sprite texture. Our representation supports such decompositions, but the calling application is in charge of computing the subsprite positioning. It is also possible to define higher order local parameterizations.

Deformation-proof texturing
Our tree structure allows to store and retrieve data for 3D locations. However, it was meant to associate these informations to an *object surface*. If the object is rotated rescaled or animated we want

the texture data to stick to the surface. This is exactly equivalent to the case of solid textures [Per85]. The classical solution is to rely on a *3D parameterization* (u,v,w) stored at each vertex and interpolated as usual for any fragment. This parameterization can be seen as the *reference* or rest state of the object and can conveniently be chosen in $[0,1]^3$. Note that the reference mesh does not need to be the same than the rendered mesh as long as they can be represented by the same N^3-tree. For instance, a subdivision surface can be subdivided further without requiring to update the texture. The (u,v,w) of newly created vertices just have to be interpolated linearly.

Since our representation defines a 3D texture, the rendered mesh does not even need to be a surface. In particular, point-based representations and particle systems can be textured conveniently. Finally, a high resolution volume can be defined by 3D sprites and sliced as usual for the rendering.

7 Blending sprites

When multiple sprites overlap, the resulting color is computed by blending together their contributions. Various ways of compositing sprites can be defined. The texturing methods that rely on multipass rendering are limited to basic frame buffer blending operations. Most often it is the transparency blending. Since our blending is performed in a fragment program we suffer no such limitations. Our model relies on a customizable blending component to blend the contributions of the overlapping sprites.

We implemented non standard blending modes such as blobby-painting (figure 1(e)) or cellular textures [Wor96] (i.e. Voronoi, figure 1(f)). The first effect corresponds to an implicit surface defined by sprite centers. The second effect selects the color defined by the closest sprite. Both rely on a distance function that can be implemented simply by using a pattern containing a radial gradient in the alpha value A (i.e. a tabulated distance function).

8 Filtering

We can distinguish three filtering cases: linear interpolation for close viewpoints (*mag filter*), MIP-mapping of sprites (*min filter*) and MIP-mapping of the N^3-tree.

Linear interpolation
Linear interpolation of the texture of each sprite is handled naturally by the texture unit of the GPU. As long as the blending equation between the sprites is linear the final color is correctly interpolated.

MIP-mapping of sprites
The min filtering is used for faces that are either distant or tilted according to the viewpoint. The MIP-mapping of the texture of each sprite can be handled naturally by the texture unit of the GPU. As long as the blending equation between the sprites is linear, filtering of the composite texture remains correct: Each sprite is filtered independently and the result of the linear blending still corresponds to the correct average color. However, since we computed the (u,v) texture coordinates instead of relying on a simple interpolation of (u,v) defined at vertices, the GPU does not know the derivatives relative to screen space and thus cannot evaluate the MIP-map level. To achieve correct filtering we

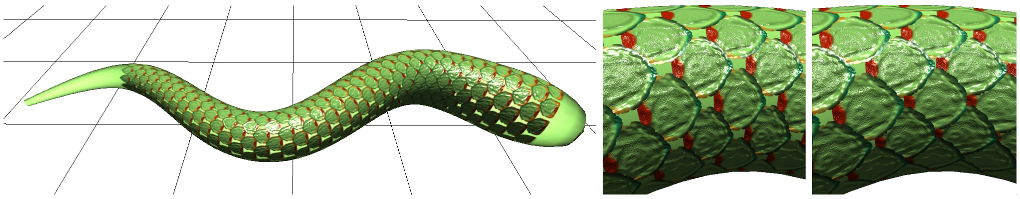

Figure 9: Undulating snake mapped with 600 overlapping texture sprites which common pattern (color+bump) have a 512×512 resolution. The virtual composite texture thus has a 30720×5120 resolution. One can see the correct filtering at sprite edges. This figure demonstrates the independent tuning of each scale size in order to simulate rigid scales. *Middle:* without stretch compensation. The texture is homogeneous and is stretched depending on the curvature. *Right:* with stretch compensation. The scales slide onto each other and overlap differently depending on the curvature (see also the video).

compute the derivatives (we rely on the `ddx` and `ddy` derivative instructions on Nvidia hardware) explicitly before accessing the texture.

MIP-mapping of the N^3-tree
If the textured object is seen from a very distant viewpoint, multiple cells of the tree may be projected into the same pixel. This is exactly equivalent to the *geometric aliasing* case we discussed in section 2.2: aliasing will occur if the cells contain different color statistics. Tree filtering can be achieved similarly to what was done in the case of [DGPR02, BD02] (i.e. defining nodes values that are the average of child values, which corresponds to a regular MIP-mapping). In our case we first need to evaluate the average color of the leaves from the part of the sprites they contain. However cell aliasing does not often occur in practice: First, the cell size does not depend on the sprite size (in particular, small sprites are stored in large cells). Second, our insertion algorithm presented in section 5.2 tends to minimize the tree depth to avoid small cells. Finally, small neighboring cells are usually covered by the same sprites and therefore have the same average color. Thus we did not need to implement MIP-mapping of the N^3-tree for our demos. Apart from very distant viewpoints (for which the linearity hypothesis assumed by every texturing approach fails), the only practical case where cell aliasing occurs is when two different sprites are close to each other and cannot be inserted inside the same leaves: The two sprites have to be separated by splitting the tree. As a result, small cells containing different sprites are generated. These cells are likely to alias if seen from a large distance.

9 Applications and Results

9.1 Examples

We have created various examples shown on figure 1, figure 9 and the accompanying video to test and illustrate our system.

Texture authoring (figure 1(c) and video)
In this example, the user interactively pastes sample textures onto a surface. After having provided

a set of texture patterns, the user can simply click on the mesh to create a texture sprite. The latter can then be interactively scaled, rotated, or moved above or below the already existing sprites. The night-sky bunny was textured in a few minutes.

This typically illustrates how an application can use our representation: Here, it is responsible for implementing the user interface, placing the sprites (a simple *picking* task), and orienting them. Requests are done to our API to insert and suppress sprites as they move.

Note that sprites can overlap, but also large surface parts can remain uncovered. This permit to use an ordinary texture (or a second layer of composite texture) on the exposed surface. In particular, this provides a way to overcome the overlapping limit by using multipass rendering.

Lapped texture approximation (figure 1(a) and video)
This example was created using the output of the *lapped texture* algorithm [PFH00] as an input to our texturing system. Our sprite-based representation fits well with the approach of this texture synthesis algorithm in which small texture elements are pasted on the mesh surface. Our representation stores such textures efficiently: The sample is stored only once at full resolution and the N^3-tree minimizes the memory space required for positioning information. Moreover, rendering does not suffer from filtering issues created by atlases or geometrical approaches (see video), and we could rely on a low resolution mesh. Since lapped texture involves many overlapping of sprites, in our current implementation we used two layers of textures to overcome Cg and hardware limitations (pixels shaders use more registers than they should).

Animated sprites (figure 1(b),(c) and video)
Sprites pasted on a 3D model can be animated in two ways.
• First, the application can modify the positioning parameters (position, orientation, scaling) at every frame, which is not time consuming. Note that it is also what a texture authoring tool would do. Particle animation can be simulated as well to move the sprites (*e.g.,* drops). In figure 1(b), the user has interactively placed gears. Then the sprites rotation angle are modified at each frame (clockwise for sprites with even id and counter-clockwise for odd ones).
• Second, the pattern bound to a sprite can cycle over a set of texture patterns, simulating an animation in a cartoon-like fashion. The patterns of figure 1.c are animated that way. Both the above examples were rendered at about 15 fps at a resolution of 512×512.

Snake scales (figure 9 and video)
As explained above each sprite can be independently scaled and rotated. This can even be done by the GPU as long as a formula is available. For illustration we emulated the behavior of rigid scales: usually the texture deforms when the underlying surface is deformed (figure 9,middle). Here we estimate the local geometric distortion and scale the sprites accordingly so as to compensate for the deformation of the animated mesh. Our example is an undulating snake covered by scales: one can see (especially on the video) that the scales keep their size and slide on each other in concave areas. Note that this has similarities with the *cellular textures* of Fleischer *et. al.* [FLCB95]: sprites have their own life and can interact. But in our case no extra geometry is added. We really define

a textural space in which the color can be determined at any surface location, so we do not have to modify the mesh to be textured.

Octree textures (figure 1(d) and video)
We have reimplemented the DeBry et. al. *octree textures* [BD02] based on our system in order to benchmark the efficiency of our GPU N^3-tree model. In this application no sprite parameter is needed, therefore we directly store color data in the leaf cells of the indirection grids. The octree texture can be created by a paint application or by a procedural tool. Here we created the nodes at high resolution by recursively subdividing the N^3-tree nodes intersecting the object surface. Then we evaluated a Perlin marble noise to set the color of each leaf. For filtering, we implemented a simple tri-linear interpolation scheme by querying the N^3-tree at 8 locations around the current fragment. We obtained a frame rate of about 20 fps at 1600×1200 screen resolution. The timings of DeBry *et. al.* [BD02] software implementation was about one minute to render a 2048×1200 still image. This proves that this approach benefits especially well from our GPU implementation.

9.2 Performance

Rendering time
Performances for the examples presented in the paper are summarized in table 1. Measurements were done on a GeForceFX 5900 Ultra. The system is entirely implemented in Nvidia Cg. The performance of our N^3-tree allows for fast rendering of color octree textures; but the complete texture sprite system usually performs at a lower – but still interactive – frame rate. The loss of performance is mainly due to the limitations of today's GPU (which might be obsolete soon). The lack of true branching in fragment programs implies that a lookup in our composite texture always involves as many computation as if O_{max} sprites were stored in the deepest leaves of the tree (another consequence is that the rendering cost remains constant independently of the number of sprites stored). We expect twice the current frame rate with true branching instructions in fragment programs. This was estimated by counting the number of pixels that would not involve a complete evaluation of the program.

Note that since the cost is in the fragment shader, the rendering cost is proportional to the number of pixels drawn: The rendering of an object that is far or partly occluded costs less; *i.e.,* you pay only for what you see.

Memory usage
Our textures require little memory in comparison to the standard textures needed to obtain the same amount of details. Our tests show that texturing the Stanford bunny with an atlas automatically generated by a modeling software (see video) would require a 2048×2048 texture (*i.e.,* 16 MB) to obtain an equivalent visual quality. Moreover, we show on the video that atlas discontinuities generate artifacts. The memory size used by our various examples is summarized in table 2. Note that since textures must have power of two dimension in video memory the allocated memory size is usually greater than the size of the structure.

	number of sprites	node size	tree depth	max. overlap	FPS 800×600
Lapped	536	4	4	16	5
Gears	50	4	2	4	20
Stars	69	4	2	8	10
Octree	none	2	9	none	40
Blobby	136	4	3	10	5
Voronoi	132	4	3	12	5

Table 1: Performances for examples of figure 1.

	used memory	allocated memory	total memory
Lapped	1 MB	1.6 MB	1.9 MB
Gears	0.012 MB	0.278 MB	0.442 MB
Stars	0.007 MB	0.285 MB	5.7 MB
Octree	16.8 MB	32.5 MB	32.5 MB
Blobby paint	0.141 MB	0.418 MB	0.434 MB
Voronoi	0.180 MB	0.538 MB	0.554 MB

Table 2: Storage requirements for examples of figure 1.

10 Conclusion and future work

We have introduced a new representation for texturing 3D models with texture patterns. This representation allows very high resolution texture using little video memory and does not suffer from filtering artifacts that numerous existing methods have. Since it defines a texture it does not require any modification of the textured geometry. In our approach, customizable texture sprites are defined and stored spatially in a N^3-tree bounding the object. The system is flexible in many ways: each sprite can be independently animated, moved, scaled and rotated. This permits to have many existing methods such as interactive painting on surfaces, fast lapped-like texture rendering, octree texture rendering, texture details, all available within the same texturing system with better quality and using less memory. Our method also tends to minimize the amount of memory used to store the complete texture since it inherently makes use of instantiation and hierarchical data-structure.

We have demonstrated how such a system is well suited for GPU implementation. Even with the limitations of today's GPUs (which will not last) making our N^3-tree queries far from optimal, we showed that good performance is already possible. If it is not sufficient for a given application, we showed on the video that it is easy to automatically create a regular texture (*e.g.,* an atlas) from a composite virtual texture by drawing it in texture space.

Future work

In this paper we have demonstrated several types of usage of our system. However, the possibilities are endless and we would like to explore other kinds of textures enabled by this sprite instantiation scheme. We also showed that relying on a spatial structure – and no surface parameterization – the rendered objects no longer require to be meshes. In particular, our approach could bring interesting new tracks to point-based and volumetric representations.

References

[BD02] BENSON D., DAVIS J.: Octree textures. In *SIGGRAPH 2002 Conf. Proc.* (2002), ACM Press, pp. 785–790.

[Bod] BODYPAINT3D: Texturing software.
http://www.maxoncomputer.com/.

[CSHD03] COHEN M. F., SHADE J., HILLER S., DEUSSEN O.: Wang tiles for image and texture generation. *SIGGRAPH 2003 Conf. Proc.* (2003), 287–294.

[Dee] DEEP PAINT3D: Texturing software.
http://www.righthemisphere.com/products/dp3d/.

[DGPR02] DEBRY D., GIBBS J., PETTY D. D., ROBINS N.: Painting and rendering textures on unparameterized models. In *SIGGRAPH 2002 Conf. Proc.* (2002), pp. 763–768.

[DMLG02] DISCHLER J., MARITAUD K., LÉVY B., GHAZANFARPOUR D.: Texture particles. *EUROGRAPHICS 2002 Conf. Proc.* (2002), 401–410.

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. *SIGGRAPH 2001 Conf. Proc.* (2001), 341–346.

[FLCB95] FLEISCHER K. W., LAIDLAW D. H., CURRIN B. L., BARR A. H.: Cellular texture generation. *Computer Graphics (SIGGRAPH '95 Conf. Proc.)* (1995), 239–248.

[HH90] HANRAHAN P., HAEBERLI P. E.: Direct WYSIWYG painting and texturing on 3D shapes. In *SIGGRAPH 90 Conf. Proc.* (1990), pp. 215–223.

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Proc. of Graphics Hardware 2002* (2002), pp. 7–15.

[KSE*03] KWATRA V., SCHÖDL A., ESSA I., TURK G., BOBICK A.: Graphcut textures: Image and video synthesis using graph cuts. *SIGGRAPH 2003 Conf. Proc.* (2003).

[LN03] LEFEBVRE S., NEYRET F.: Pattern based procedural textures. In *I3D 2003 Conf. Proc.* (2003), pp. 203–212.

[Mil00] MILLER N.: Decals explained.
http://www.flipcode.com/tutorials/tut_decals.shtml.

[MYV93] MAILLOT J., YAHIA H., VERROUST A.: Interactive texture mapping. In *SIGGRAPH 93 Conf. Proc.* (1993), pp. 27–34.

[NC99] NEYRET F., CANI M.-P.: Pattern-based texturing revisited. In *SIGGRAPH 99 Conf. Proc.* (1999), pp. 235–242.

[NHS02] NEYRET F., HEISS R., SENEGAS F.: Realistic Rendering of an Organ Surface in Real-Time for Laparoscopic Surgery Simulation. *The Visual Computer 18* (2002), 135–149.

[Pea85] PEACHEY D. R.: Solid texturing of complex surfaces. In *SIGGRAPH 85 Conf. Proc.* (1985), pp. 279–286.

[Per85] PERLIN K.: An image synthesizer. In *SIGGRAPH '85 Proceedings* (1985), pp. 287–296.

[PFH00] PRAUN E., FINKELSTEIN A., HOPPE H.: Lapped textures. In *SIGGRAPH 2000 Conf. Proc.* (2000), pp. 465–470.

[SCA02] SOLER C., CANI M.-P., ANGELIDIS A.: Hierarchical pattern mapping. *SIGGRAPH 2002 Conf. Proc.* (2002), 673–680.

[Tex] TEXTURE PAINT: Texturing software.
http://home.t-online.de/home/uwe_maurer/texpaint.htm.

[Tur01] TURK G.: Texture synthesis on surfaces. In *SIGGRAPH 2001 Conf. Proc.* (2001), pp. 347–354.

[WL00] WEI L.-Y., LEVOY M.: Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH 2000 Conf. Proc.* (2000), Akeley K., (Ed.), ACM Press, pp. 479–488.

[WL01] WEI L.-Y., LEVOY M.: Texture synthesis over arbitrary manifold surfaces. In *SIGGRAPH 2001 Conf. Proc.* (2001), pp. 355–360.

[Wor96] WORLEY S. P.: A cellular texture basis function. In *SIGGRAPH 96 Conf. Proc* (1996), pp. 291–294.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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