Texture Sprites: Texture Elements Splatted on Surfaces

Sylvain Lefebvre* Samuel Hornus* Fabrice Neyret*
GRAVIR / IMAG-INRIA

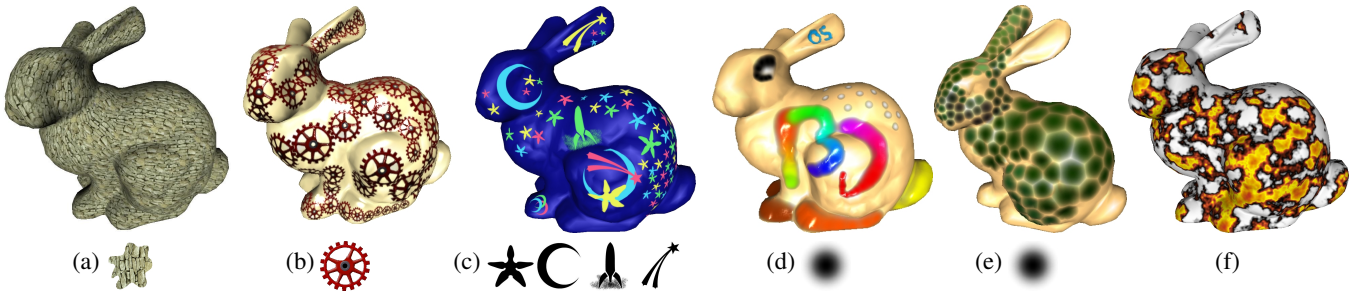

Figure 1: *From left to right*: (a) Lapped textures. (b-c) Animated sprites. (d) Blobby painting. (e) Voronoi blending of sprites. (f) Octree texture. The bottom line shows the texture *patterns* used (which are the only user defined textures stored in video memory). All these bunnies are rendered in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

Abstract

We present a new interactive method to texture complex geometries at very high resolution, while using little memory and without the need for a global planar parameterization. We rely on small texture elements, the *texture sprites*, locally splatted onto the surface to define a composite texture. The sprites can be arbitrarily blended to create complex surface appearances. Their attributes (position, size, texture id) can be dynamically updated, thus providing a convenient framework for interactive editing and animated textures. We demonstrate the flexibility of our method by creating new surface aspects difficult to achieve with other methods.

Each sprite is described by a small set of attributes which is stored in a hierarchical structure surrounding the object's surface. The patterns supported by the sprites are stored only once. The whole data structure is compactly encoded into GPU memory. At run time, it is accessed by a fragment program which computes the final appearance of a surface point from all the sprites covering it. The overall memory cost of the structure is very low compared to the resulting texturing resolutions. Rendering is done in real–time. The resulting texture is linearly interpolated and filtered.

CR Categories: I.3.7 [Computer Graphics]: Three -Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: texturing, texture sprites, octree textures, decals, real–time rendering, graphics hardware

*email: FirstName.Name@imag.fr
URL: http://www-evasion.imag.fr/Publications/2005/LHN05

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, texturing 3D models with very high resolutions creates two major difficulties. First, the storage cost of high resolution texture maps can easily exceed the available texture memory. The difficulty to create planar parameterizations further worsens this problem since memory can be wasted by unused space in the texture maps or by distorted areas. Second, creating high resolution textures proves to be a difficult and tedious task.

A current trend in computer graphics is to synthesize large textures from image samples [De Bonet 1997; Wei and Levoy 2000]. Indeed, many surfaces have an homogeneous appearance which can be well captured by a small sample. Many of these algorithms produce a larger texture by combining patches taken from the texture sample [Efros and Freeman 2001; Kwatra et al. 2003]. However, because of the lack of efficient representation of patch–based textures, they often explicitly store the resulting texture in a large image, which wastes memory. Another approach is to introduce new geometry to position the patches directly onto the surface [Praun et al. 2000; Dischler et al. 2002]. If this approach greatly reduces texture memory consumption, it also increases the geometrical cost, for texturing purposes only. Moreover, this hinders geometric optimizations such as triangle strips and geometric level of details.

Besides homogeneous appearances, textures are also a solution to encode scattered objects like footprints, bullet impacts [Miller 2000], or drops [Neyret et al. 2002; Lefebvre 2003]. These are likely to appear dynamically during a video game. In this situation also, the lack of representation for textures composed of sparse elements creates difficulties. Storing scattered details in a large texture covering the mesh wastes a lot of memory, since the texture then contains large empty areas. The common solution is to use *decals* instead, i.e., to put the texture elements on extra small textured transparent quads. However, such marks do not stick correctly to curved surfaces and intersection between decals yields various artifacts such as discontinuities and flickering due to Z–fighting. All 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 composed of various texture elements. Since the texture elements live on the object

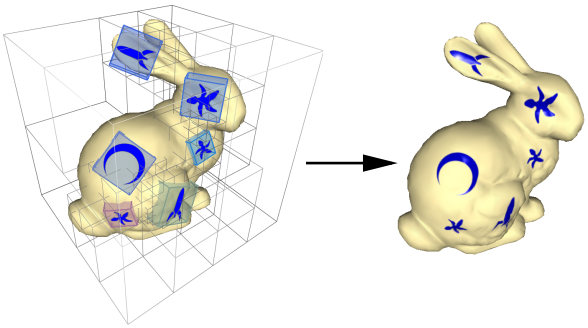

Figure 2: The attributes of the sprites are stored in an octree structure surrounding the mesh surface.

surface and can be updated dynamically we refer to them as *texture sprites*. Our representation provides a general and convenient scheme for sprite-based texturing, ranging from homogeneous appearances obtained by overlapping many sprites, to the interactive addition of local details. It does not require the mesh to conform to any constraint. In particular it does not require to compute a global planar parameterization, nor to modify the initial mesh: The geometry and the composite texture are independent.

The contributions of this paper are:

- A parameterization-free texture representation allowing *efficient storage* and *filtered rendering* of composite textures.
- An efficient solution to dynamic decal management and animated texture elements.
- New appealing texturing effects made possible by our texture representation, as illustrated in Figures 1 and 9.
- A complete GPU-implementation that makes our method usable both in texture authoring tools and real-time applications.

Our key idea is to store the mapping parameters of the sprites along the mesh surface, using a hierarchical structure similar to an octree (see Figure 2). At run–time, the renderer determines which sprites are covering the rasterized pixel, and computes the final color by combining the sprite's textures. Customizable blending operators allow to treat overlapping sprites according to the user requirements. The resulting composite texture is accessed from a fragment program using 3D texture coordinates.

2 Previous work and texturing issues

2.1 High resolution textures

When creating detailed textures, one has to cope with two issues: authoring the texture content and making it small enough to fit into texture memory.

A solution to save memory while keeping high resolution is to rely on repeated patterns using tiles (see Figure 3). Games usually use quad tiles for terrain textures. Recently, Cohen *et al.* [2003] proposed an approach to avoid regularity with a small set of precomputed tiles. This approach was ported to the GPU by Wei [2004]. Lefebvre and Neyret [2003] also proposed a similar method to dynamically instantiate texture patterns at arbitrary locations in a 2D texture space. Arbitrarily large textures are thus created at low memory cost. These approaches do not require modifications of the mesh geometry. However, this process does not transfer well to arbitrary surfaces where the parameterization introduces distortions on the tiles. Therefore, these methods cannot represent composite textures on arbitrary surfaces and suffers from the same artifacts as traditional mapping on curved geometry.

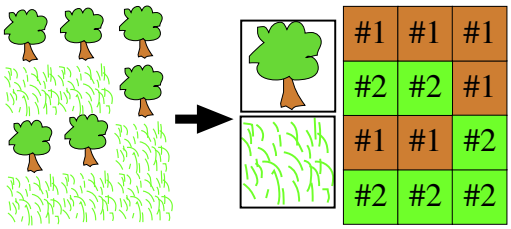

Figure 3: A *tiling* saves memory by instancing a small set of patterns.

Another convenient approach to produce highly detailed textures is to automatically synthesize large textures from an image sample [De Bonet 1997; Wei and Levoy 2000; Efros and Freeman 2001; Kwatra et al. 2003]. The result is stored in a 2D texture map: A parameterization of the object is mandatory. The distortions and discontinuities often introduced by the parameterization may lead to visual artifacts. In particular, it may require to increase the texture size to avoid loss of resolution in distorted areas. Since the synthesized texture is also composed of similar patterns the representation is not memory efficient. Other algorithms proceed directly on the surface to avoid parameterization [Praun et al. 2000; Turk 2001; Wei and Levoy 2001; Soler et al. 2002; Dischler et al. 2002]. To store the result they often modify the input geometry, which breaks the independence between the texture and the mesh. This can produce an overly sampled geometry and hinders the use of geometric level of details or the dynamic update of the texture content. Authors proposing better solutions for the storage either generate a soup of small textured polygons [Dischler et al. 2002] or redraw several times the faces on which several texture elements are mapped [Praun et al. 2000]. As explained in this last paper, instancing instead of duplicating texture elements allows one to obtain a far better texture resolution with less memory. However, drawing several superimposed faces increases the geometry processing cost and yields various blending issues and Z–buffer conflicts. Moreover, the dynamic creation and update of these small geometry patches is difficult when geometry is complex or animated.

2.2 Filtering issues

It is important to realize that GPU rendered scenes are anti-aliased mostly thanks to texture filtering mechanisms and despite aliasing in the rasterized geometry. However, this works only under the assumption that the texture mapping function is continuous along the surface. Each time a mapping discontinuity is defined on the surface the filtering is no longer correct, for both linear interpolation and MIP-mapping. Note that oversampling decreases but does not remove these artifacts.

The new powerful features of GPUs also yield 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, estimating correctly filtered value is no longer possible using an embedded mechanism. It is the programmer's responsibility to handle filtering. Even so, detecting and fixing the problem is not straightforward [Kraus and Ertl 2002; Lefebvre and Neyret 2003; Wei 2004].

3 Sprite-Based Textures

This section describes our texturing method dedicated to the editing and rendering of sprite-based textures. Sprites are local texture elements that are dynamically projected onto the surface.

Our key idea is to store the sprite attributes (position, orientation, etc) in a 3D hierarchical structure surrounding the mesh. Until

recently, volumetric representations have been used only for solid texturing [Perlin 1985; Peachey 1985; Worley 1996] and volume rendering. We follow the idea of *octree textures* [DeBry et al. 2002; Benson and Davis 2002] which relies on an octree to efficiently store color information along *a surface*. However, instead of storing colors we store the sprites attributes in the leaves of the hierarchical grid (see Figure 2). The entire data-structure is compact: Many sprites share a same texture pattern and information is stored only around the surface.

The structure is encoded into GPU texture memory and is accessed from a fragment program using 3D texture coordinates. Thus, the original mesh geometry is not modified and can be optimized for rendering (triangle strips, level of details) independently from the texture. The composite texture (composed of all the sprites) is linearly interpolated and MIP-mapped.

Sprite-based textures are easy to author: The user clicks on the surface to add sprites, drags and drops them and he interactively changes their size or orientation. The system is very flexible: Sprite ordering is controllable and overlapping sprites can be mixed together to achieve various appealing effects (Figure 1). Each sprite can be independently animated, for instance to react to surface deformations as shown in Figure 9.

Our representation is interesting for high quality software renderers, to reduce memory requirements of composite textures and to avoid rendering artifacts. Since it is amenable to GPU implementation despite the more constrained context, we focus on the description of a GPU implementation. It is easy to adapt it to a software framework.

Several difficulties have to be solved: Section 4 describes how to manage the hierarchical 3D grid on the GPU so that it can be accessed from a fragment program. Section 5 shows how we use the hierarchical grid to store and retrieve sprites attributes. Section 6 explains how we filter the composite texture and blend overlapping sprites together. Section 7 presents various appealing texturing effects made possible by our composite textures, as well as performance results. We conclude and discuss future work in Section 8.

4 Implementing a 3D hierarchical grid on a GPU

We now explain how to store the hierarchical grid in texture memory, and how to retrieve data from the grid at a given 3D location. This is implemented in a fragment program (or pixel shader) executed per-pixel on the 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, it is crucial to think about the consequences of each choice in terms of performance, and precision issues complicate the task (these include low dynamics of values and the fact that indices are floating points). The reader can also refer to [Lefebvre et al. 2005] for details on how to implement and use octree textures on the GPU.

[Kraus and Ertl 2002] 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 relevant texture data in texture memory. 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 containing s tiles and an indirection map I of resolution n is used to recover the pixel color of any given location: $T(x) = P\left(\frac{I(x)+frac(x\cdot n)}{s}\right)$ (where $frac(x)$ denotes the fractional part of x).

We extend these two ideas to create a *hierarchical* 3D grid storing sprite positioning information. We call this structure a N^3**-tree** because each internal node has N^3 children (N in each dimension). An octree corresponds to $N = 2$ (in our implementation we use $N = 4$). See Section 4.3 for a discussion on the choice of N.

4.1 N^3-tree memory layout

Tree structure

Figure 4 shows the grid structure: Each node represents an *indirection grid* consisting of N^3 cells; the cell content (whose type is determined by a flag stored in the A-channel) is either *data* if it is a leaf ($A = 1$), or a *pointer* to another node ($A = \frac{1}{2}$). The 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 does vectorial operations on 4D vectors. In the following we use vector notations (operations are done component by component).

Notations

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

Nodes are packed in the indirection pool (see Figure 4-right). Let S_u, S_v and S_w be the number of allocated nodes in the u, v and w directions. We define $S = (S_u, S_v, S_w)$ as the size of the indirection pool using the vectorial notation. The resolution of the texture hosting the indirection pool is therefore $N\,S_u \times N\,S_v \times N\,S_w$. 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 $P_i = \frac{\lfloor P \cdot S \rfloor}{S}$ and $P_f = \frac{frac(P \cdot S)}{S}$. P_i identifies the node which is stored in the area $[P_i, P_i + \frac{1}{S})$ of the indirection pool. P_i is called the *node coordinates*. P_f is the local coordinate of P within the node's 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 $M_{l,f}\,N^l = P_{l,f}\,S$. Thus:

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

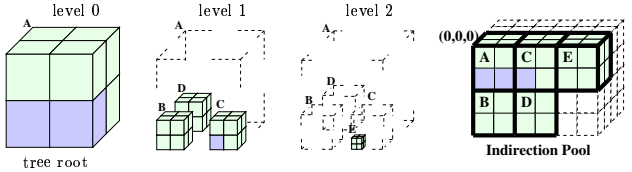

Figure 4: An example tree. The five nodes A,B,C,D and E are packed in the indirection pool (*right*). In this example each node is an indirection grid of size 2^3 ($N = 2$). The cells of the indirection grids contain either data (*light, green cells*) or an indirection (i.e. a pointer) to a child node (*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. For rendering, we need to retrieve the data corresponding to a given location M in space (more precisely, on the object surface). This is done per-pixel in a fragment program.

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 Equation (1). This 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 = <0,0,0,0>;
for (float i=0; i<TREE_MAX_DEPTH; i++){
  // child location
  float3 P = (frac(M)+rgba.xyz*255.0)*inv_S;
  rgba = (float4)tex3D(PoolTex,P); // access indir. pool
  if (rgba.w > 0.9) break; // type of node = leaf
  if (rgba.w < 0.1) discard; // type of node = empty
  M=M*N; // go to next level
 }
return rgba;
```

Figure 5: The pseudo–code of our tree lookup. M is provided by the CPU via the 3D texture coordinates. `inv_S`$= \frac{1}{S}$. As the `break` statement is not available on all hardware 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 $S \cdot P_{l,i}$ instead of $P_{l,i}$ in the indirection cells. Recall that $P_{l,i} \in [0,1)$ is the coordinate of a node's indirection grid within the indirection pool. Therefore, $S \cdot P_{l,i}$ is an integer, and we can store it as such, which has two advantages. First, it optimizes the encoding: We can choose the number of bits to be stored in the *RGB* channels to exactly represent at most S indices. Second, the values of the pointers do not depend on 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 internal registers 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 N and S. Recall that S controls the size of the indirection pool and thus the maximum number of nodes that can compose the N^3-tree. Writing $N = 2^{e_N}$, $S = (2^{e_S}, 2^{e_S}, 2^{e_S})$ and using standard floating point values with a mantissa of 23 bits, the deepest reachable depth is $d_{max} = \lfloor \frac{23-e_S}{e_N} \rfloor$ (see Figure 6).

Increasing the size N of the tree nodes allows to store more information at a given tree depth. Usually for a same set of sprites this reduces the tree depth resulting in better performance (fewer texture lookups are required to go down the tree). An application limited by storage would rather choose a small N value to pack texture data as tightly as possible. However, for applications where rendering performance is crucial a larger value of N would be preferable to limit per-fragment computations while still offering high resolution in deepest tree levels.

N	S (maximum number of nodes)	d_{max}	max. reachable resolution
2	$16 \times 16 \times 16$	19	$(2^{19})^3$
4	$16 \times 16 \times 16$	9	$(2^{18})^3$
8	$16 \times 16 \times 16$	6	$(2^{18})^3$
2	$32 \times 32 \times 32$	18	$(2^{18})^3$
4	$32 \times 32 \times 32$	9	$(2^{18})^3$
2	$128 \times 128 \times 128$	16	65536^3
4	$128 \times 128 \times 128$	8	65536^3

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

Increasing the maximum number of nodes S that can be stored in the indirection pool allows to create larger structures. An increase of S reduces d_{max}, but only by a small amount.

5 Managing the texture sprites

Each sprite is associated with various parameters (including positioning parameters described in Section 5.3) 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 detect whether two sprites overlap.

5.1 Sprite storage

Storing sprite parameters
The parameters associated with a sprite 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 this set directly in each leaf: This would waste memory and complicate the update of sprites. Instead we introduce an indirection: The *sprite parameter table* is a texture containing all the parameters for all the sprites. Each sprite corresponds to an index into this texture; each texel encodes a parameter set. 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 texture 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 must be copied.

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* (see Figure 7). 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 leaf.

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

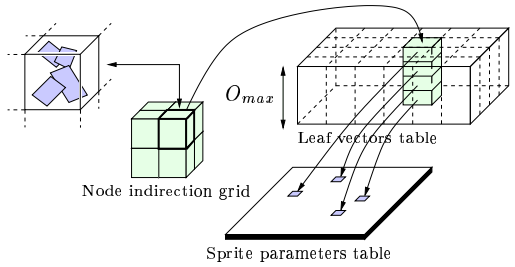

Figure 7: Each indirection cell stores the index of the corresponding leaf vector. Each cell of the leaf vector stores the indices of the sprite parameters.

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 locally generate a set of small nodes 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.

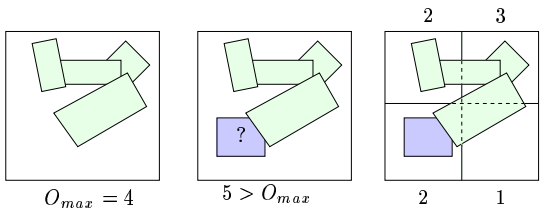

Figure 8: A new sprite (*dark, blue*) is inserted in a full leaf. As the sprite does not overlap with all the sprites (*light, green*), there is a level of subdivision at which the new sprite can be inserted. Subdividing lets us keep the sprite count less than or equal to O_{max}.

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 some sprite 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).

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, simply by ordering the sprites in the leaf vectors table.

5.3 Positioning the sprites on the surface

At this stage we can know which sprites are in each cell surrounding the object surface, but no more. In this section we describe how to encode and recover their precise position 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 back to texture space, onto the sprite.

We associate a frame to each sprite and use a parallel projection to the surface. The positioning parameters are:

- a center point **c** (sprite handle),
- two vectors $\mathbf{l}, \mathbf{h}$ defining the plane, scale and orientation of the pattern,
- a normal vector **n** defining the direction of projection.

Let M denote the matrix $(\mathbf{l}, \mathbf{h}, \mathbf{n})$. Once the sprite parameters are fetched for a given point P, we compute in the pixel shader $U = M^{-1} \cdot (P - \mathbf{c})$ 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**. This solves the ambiguous color assignment with thin or flat two-sided objects described in [DeBry et al. 2002; Benson and Davis 2002].

The sprite projection on the surface is equivalent to defining a local linear parameterization. If the distortion is too high for a given sprite, when a geometric feature is smaller than the sprite, it is possible to split it in multiple sub–sprites to minimize distortion. Each sub–sprite displays only a sub–region of the initial sprite texture. Our representation supports such decompositions, but the calling application is in charge of computing the sub–sprite positioning. In practice, to avoid visible projection artifacts we attenuate the sprite contribution according to the angle between the sprite and surface normals.

Deformation-proof texturing
Our tree structure allows to store and retrieve data from 3D locations. However, it is 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 [Perlin 1985]. The usual 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 as the rendered mesh as long as they can be textured by the same N^3-tree. For instance, a subdivision surface can be subdivided further without requiring one 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 could even be defined by 3D sprites and sliced as usual for volume rendering.

5.4 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. In most cases, it is transparency blending. Since our blending is performed in a fragment program, we do not suffer from 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 [Worley 1996] (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).

6 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 standard texture units 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 explicitly compute the (u,v) texture coordinates within the fragment program, the GPU does not know the derivatives relative to screen space and thus cannot evaluate the MIP-map level. To achieve correct filtering we compute the derivatives explicitly before accessing the texture. (We rely on the `ddx` and `ddy` derivative instructions of the HLSL or Cg languages).

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. Aliasing will occur if the cells contain different color statistics. Tree filtering can be achieved similarly to what was done in the case of [DeBry et al. 2002; Benson and Davis 2002] (i.e., defining nodes values that are the average of child values, which corresponds to a standard MIP-mapping). In our case we first need to evaluate the average color of the leaves from the portion of the sprites they contain. However, cell aliasing does not occur often 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 leaf. 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.

7 Applications and Results

7.1 Examples

We have created various examples to illustrate our system, shown on Figure 1, Figure 9 and in our video (available at `http://www-evasion.imag.fr/Publications/2005/LHN05`).

Texture authoring (Figure 1(c) and video)
In this example, the user interactively pastes texture elements 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, the application is responsible for implementing the user interface, placing the sprites (a simple *picking* task), and orienting them. Requests are sent to our texture sprites API to delete and insert sprites as they move.

Note that sprites can overlap, but also large surface parts can remain uncovered. This permits the use of 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 Textures* algorithm [Praun et al. 2000] 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 use the initial low resolution mesh. Since lapped texture involves many overlapping of sprites, in our current implementation we use two separate composite textures to overcome hardware limitations (the maximum number of registers and instructions limit the maximum number of overlapping sprites).

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. 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 is 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 this way. See Table 1 for frame rate measurements.

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, and as long as the sprite bounding volume remains unchanged. For illustration we emulated the behavior of rigid scales: usually the texture deforms when the underlying surface is deformed (Figure 9, middle). We estimate the local geometric distortion and scale the sprites accordingly 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

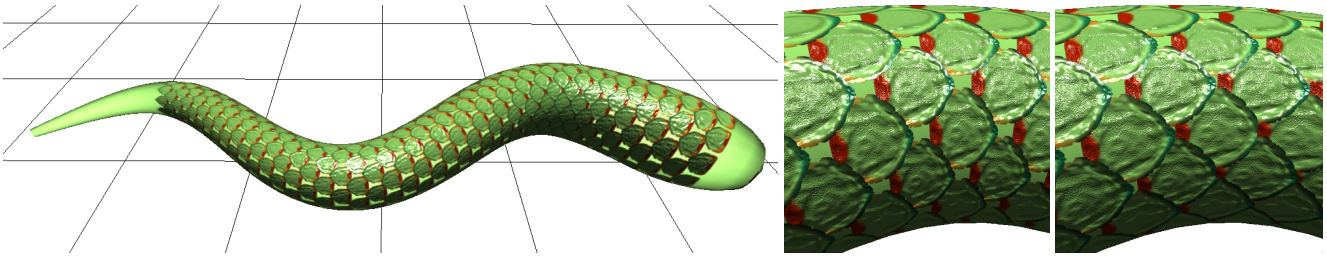

Figure 9: Undulating snake mapped with 600 overlapping texture sprites whose common pattern (color+bump) have a 512×512 resolution. The virtual composite texture thus has a 30720×5120 resolution. One can see the correct filtering at sprite edges. This figure demonstrates the independent tuning of each scale aspect-ratio in order to simulate rigid scales. *Middle:* Without stretch compensation. The texture 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).

et al. [Fleischer et al. 1995]: Sprites have their own life and can interact. But in our case no extra geometry is added: Everything occurs in texture space. 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(f) and video)
We have reimplemented the DeBry *et al. octree textures* [DeBry et al. 2002] with 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. The bunny model of Figure 1(f) is textured with an octree texture of depth 9 (maximal resolution of 512^3). We obtain about 33 fps at 1600×1200 screen resolution, displaying the bunny model with the same viewpoint than Figure 1. The timings of DeBry *et al.* [2002] software implementation was about one minute to render a 2048×1200 still image on a 1Ghz Pentium III. This proves that this approach benefits especially well from our GPU implementation.

7.2 Performance

Rendering time
Performance for the examples presented in the paper are summarized in Table 1. Measurements were done on a GeForceFX 6800 GT, without using the dynamic branching feature. 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 frame rate. The main bottleneck comes from the maximum number of overlapping sprites allowed. Note that we did not try to optimize GPU register usage: The code is directly compiled using the Cg compiler (v1.3).

On hardware without true branching in fragment programs, 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).

On hardware allowing branching we are in a favorable case. Indeed, the tree leaves enclose large surface areas and thus neighboring pixels are likely to follow the same branching path.

Note that since the cost is in the fragment shader, the rendering cost is mostly 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.

	number of sprites	node size	tree depth	max. overlap	FPS 800×600
Lapped	536	4	4	16	27
Gears	50	4	2	8	80
Stars	69	4	2	8	26
Octree (nearest mode)	none	2	9	none	365
Blobby	125	4	3	10	70
Voronoi	132	4	3	12	56

Table 1: Performance for examples of Figure 1.

Memory usage
Our textures require little memory in comparison to the standard textures needed to obtain the same amount of detail. Our tests show that texturing the Stanford bunny with an atlas automatically generated by 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 how 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 dimensions in video memory the allocated memory size is usually greater than the size of the structure. The last column of Table 2 includes the size of the 2D texture patterns used for the demonstrated application.

	size of the structure	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.

8 Conclusions and future work

We have introduced a new representation to texture 3D models with composite textures. The final appearance is defined by the blending of overlapping texture elements, the *texture sprites*, locally applied onto the surface. We thus reach very high texturing resolution at low memory cost, and without the need for a global planar parameterization. The sprite's attributes are efficiently stored in a hierarchical grid surrounding the object's surface. Since this truly defines a 3D texture sampled per–pixel, no modification of the textured geometry is required.

The system is flexible in many ways: Each sprite can be independently animated, moved, scaled and rotated. This offers natural support for many existing methods such as interactive painting on surfaces, lapped textures rendering, dynamic addition of local details, all available within the same texturing system with better quality and using less memory. We also showed how our new representation can be used to create new texturing effects, such as animated textures and textures reacting to mesh deformations.

We described a complete GPU implementation of our texturing method, which achieves real–time performance. Moreover, if the performance are not good enough for a given application, the composite texture can be baked into a standard 2D texture using an existing parameterization (as shown in the video).

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. In particular, approaches like painterly rendering [Meier 1996] could probably benefit from our texture representation. Among the possible improvements, we would like to define sprite projector functions better than simple planar mapping in order to minimize distortion. We also showed that relying on a spatial structure – and no surface parameterization – the textured objects are no longer required to be meshes. In particular, our approach could prove interesting for point-based and volumetric representations.

9 Acknowledgments

We would like to thank John Hugues, Laks Raghupathi, Adrien Treuille, and Marie–Paule Cani for proof–reading an early version of this paper. Thanks to Emil Praun and Hugues Hoppe for providing us with the Lapped Textures result used in Figure 1, and to Nvidia for providing us with the GeForce 6800 used for this work. Also, many thanks to Laure Heïgéas and Gilles Debunne for their help in creating the accompanying video, and to our reviewers for their help in improving this paper.

References

BENSON, D., AND DAVIS, J. 2002. Octree textures. In *Proceedings of ACM SIGGRAPH 2002*, 785–790.

COHEN, M. F., SHADE, J., HILLER, S., AND DEUSSEN, O. 2003. Wang tiles for image and texture generation. In *Proceedings of ACM SIGGRAPH 2003*, 287–294.

DE BONET, J. S. 1997. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of ACM SIGGRAPH 1997*, 361–368.

DEBRY, D., GIBBS, J., PETTY, D. D., AND ROBINS, N. 2002. Painting and rendering textures on unparameterized models. In *Proceedings of ACM SIGGRAPH 2002*, 763–768.

DISCHLER, J., MARITAUD, K., LÉVY, B., AND GHAZANFARPOUR, D. 2002. Texture particles. In *Proceedings of the Eurographics Conference 2002*, 401–410.

EFROS, A. A., AND FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. In *Proceedings of ACM SIGGRAPH 2001*, 341–346.

FLEISCHER, K. W., LAIDLAW, D. H., CURRIN, B. L., AND BARR, A. H. 1995. Cellular texture generation. In *Proceedings of ACM SIGGRAPH 1995*, 239–248.

KRAUS, M., AND ERTL, T. 2002. Adaptive Texture Maps. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware 2002*, 7–15.

KWATRA, V., SCHÖDL, A., ESSA, I., TURK, G., AND BOBICK, A. 2003. Graphcut textures: Image and video synthesis using graph cuts. In *Proceedings of ACM SIGGRAPH 2003*.

LEFEBVRE, S., AND NEYRET, F. 2003. Pattern based procedural textures. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics 2003*, 203–212.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. *GPU Gems II: Programming Techniques for High–Performance Graphics and General–Purpose Computation*. Addison–Wesley, ch. Octree Textures on the GPU. ISBN 0-32133-559-7.

LEFEBVRE, S. 2003. *ShaderX2: Shader Programming Tips & Tricks*. Wordware Publishing, ch. Drops of water texture sprites, 190–206. ISBN 1-55622-988-7.

MEIER, B. J. 1996. Painterly rendering for animation. In *Proceedings of ACM SIGGRAPH 1996*, 477–484.

MILLER, N. 2000. Decals explained. `http://www.flipcode.com/articles/article_decals.shtml`.

NEYRET, F., HEISS, R., AND SENEGAS, F. 2002. Realistic Rendering of an Organ Surface in Real-Time for Laparoscopic Surgery Simulation. *The Visual Computer 18*, 3, 135–149.

PEACHEY, D. R. 1985. Solid texturing of complex surfaces. In *Proceedings of ACM SIGGRAPH 1985*, 279–286.

PERLIN, K. 1985. An image synthesizer. In *Proceedings of ACM SIGGRAPH 1985*, 287–296.

PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. 2000. Lapped textures. In *Proceedings of ACM SIGGRAPH 2000*, 465–470.

SOLER, C., CANI, M.-P., AND ANGELIDIS, A. 2002. Hierarchical pattern mapping. In *Proceedings of ACM SIGGRAPH 2002*, 673–680.

TURK, G. 2001. Texture synthesis on surfaces. In *Proceedings of ACM SIGGRAPH 2001*, 347–354.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of ACM SIGGRAPH 2000*, 479–488.

WEI, L.-Y., AND LEVOY, M. 2001. Texture synthesis over arbitrary manifold surfaces. In *Proceedings of ACM SIGGRAPH 2001*, 355–360.

WEI, L.-Y. 2004. Tile-based texture mapping on graphics hardware. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware 2004*, 55–64.

WORLEY, S. P. 1996. A cellular texturing basis function. In *Proceedings of ACM SIGGRAPH 1996*, 291–294.