Pattern Based Procedural Textures

Sylvain Lefebvre Fabrice Neyret

iMAGIS*/ GRAVIR-IMAG

Abstract

Numerous real-time applications such computer games or flight simulators require non-repetitive high-resolution texturing on large landscapes. We propose an algorithm which procedurally determines the texture value at any surface location by aperiodically combining provided patterns according to user-defined controls such as a probability distribution (possibly non stationary). Our algorithm can be implemented on programmable hardware by taking advantage of the texture indirection ability of recent graphics boards. We use explicit and virtual indirection tables to determine the pattern to apply at each pixel as well as its attributes (displacement, scaling, time...). This provides the programmer with a very high resolution virtual texture with nice properties: Low memory consumption, no periodicity, control of the statistics, numerous control parameters (which can be edited on the fly)... Our representation consists of building blocks that we combine in order to illustrate various convenient texture modalities such as aperiodic tiling, sparse convolution, domain transitions and animated textures.

Keywords: textures, proceduralism, landscape, graphics hardware

1 Introduction

Textures are an efficient representation to enhance large scenes with details. Texturing works especially well for landscapes since they can easily be parameterized. However, texturing landscapes is still a demanding problem. A user can see at the same time a detailed foreground and a wide background, pattern similarities but no regularity, some dense and some sparse areas,.... Games and flight simulators rely on various techniques to encode specified features (e.g. forests, paths, flowers) and to avoid repetitiveness while keeping the memory consumption low. Yet, these applications do not totally succeed in these tasks, thus showing visual artifacts like aliasing and regularity while still consuming a large amount of texture memory. Additionally, they often add edges to the mesh to be able to tile alternate patterns, which unnecessarily increases the mesh complexity.

We propose a procedural algorithm able to simulate a large high-resolution texture: The required memory is mainly determined by the reference patterns; the procedural algorithm is in charge of breaking the regularity without introducing constraints on the mesh since all problems are solved in texture space (i.e. a single quad could be used). This method applies to textures in its most generic meaning, comprising color, transparency (billboards, volumes), normals, etc. We show that we can also deal with animated textures. Our method is designed for programmable graphics hardware and real-time applications but can also be implemented in a software renderer.

Our paper is structured as follows: In section 2 we review the previous texturing approaches and we discuss their properties and limitations. In section 3 we describe the principle of our representation as well as its basic building blocks. Then we show in section 4 how to combine them to deal with various textures modalities. We discuss results in section 6 and future work in section 7.

2 Previous work

We give an overview of different texturing mechanisms that are related to our work. At the rendering stage there are three kinds of textures suitable for large landscapes: Pattern based textures, large unique explicit textures and procedural textures.

• Pattern based texturing relies on a library of several different smaller texture patches defining a pattern. These patterns (usually square) can offer a high local resolution. The problem is to tile these patterns while avoiding the periodicity and repetitiveness of the naive tiling. Several approaches have been proposed:

- Aperiodic tiling [Stam 1997] determines the pattern to be used in each grid cell so that no periodicity occurs while insuring continuity across tile boundaries. Games usually follow a simpler way, letting the designer encode which pattern to use at all locations of the mesh, and relying on universally matchable patterns (left-right and bottom-up edges of all patterns match).
- Triangular patterns [Neyret and Cani 1999] help breaking the periodicity, and also cope well with mapping distortions and poles. They avoid the use of any global parameterization.
- Virtual atlases [Soler et al. 2002] cover a surface by letting each face pick an area in texture space so that the global result looks continuous.
- Sparse convolution [Lewis 1989; Ebert et al. 1994] distributes the pattern locations on a random basis.

Note that all of these methods introduce constraints on the mesh: Tiling works only on quads; an atlas needs a dense mesh to sample texture coordinates correctly. Moreover, these methods do not all allow the control of local variations (either explicitly or through statistical properties).

• A large unique explicit texture avoids most of the aforementioned problems since it can be painted in order to incorporate all the desired properties. Additionally, it can be pre-distorted in order to cancel out the mapping distortions. Unfortunately, it requires a huge amount of texture memory even by today's standards. Therefore, this method cannot be used to provide a high resolution texture for a large landscape. Some hardware extensions have been introduced to reduce the memory consumption:

- Clipmaps [Tanner et al. 1998], especially thought for landscapes, offer a virtual memory management for textures allowing the usage of a texture that does not completely fit into texture memory. Still, this texture has to be designed and stored.
- Lossy texture compression is supported in almost all graphics cards [SGI n. d.]. However, the compression rate is too low to really allow for very large textures, and the quality loss is not always acceptable.

*iMAGIS is a joint project of CNRS, INRIA, Institut National Polytechnique de Grenoble and Université Joseph Fourier.
E-mail: [Sylvain.Lefebvre|Fabrice.Neyret]@imag.fr
WWW: http://www-imagis.imag.fr/Membres/Sylvain.Lefebvre/pattern

- Recently, empty space compression and variable resolution methods [Kraus and Ertl 2002; Cool 2002] have been introduced. They take advantage of the new texture indirection features of recent graphics hardware. The idea is to pack the non-blank data in a first texture, and to index it through a second texture considered as a grid, the cells of which point on the pattern to be used. This basic idea is inspiring, but a lot more can be done: in these papers the grid texture is explicit and therefore still requires a lot of memory in the case of landscapes. Moreover, quantization of the data is required to keep a low number of patterns to be stored in the reference texture. The instantiation mechanism is merely used to factor the blank tiles, and texture filtering is an issue.

• Procedural textures [Perlin 1985; Worley 1996; Ebert et al. 1994] is a very convenient mechanism to generate details at arbitrary resolution with no periodicity and very low memory. The advanced programmability of recent graphics boards promises that procedural textures implemented in hardware will soon offer almost the same flexibility than software shaders. Moreover, interesting mechanisms have been introduced to generate large virtual grids of random values with arbitrary dimensions. Still, not all kinds of material aspects can be generated using these techniques, and the calculation cost is non-negligible.

From the previous methods we wish to keep the idea of having reference pattern, the *no-constraint-on-mesh* property of large unique textures, the hashing mechanism of procedural textures, and the compact representation of the hardware-enhanced indirection textures proposed by [Kraus and Ertl 2002] and [Cool 2002].

3 Our representation

In our system a user can compose a very large texture by placing instances of reference patterns either explicitly or procedurally. Thus the user provides the reference patterns, determines the size of the virtual texture to be generated, describes the combination of actions he wishes, and provides the various maps and parameters needed to control and tune these actions.

The principle of our algorithm is to choose a pattern for given texture coordinates u, v on the fly. We subdivide the complete texture space into a virtual grid. For given u, v texture coordinates we find the corresponding cell in the virtual grid. Then we choose a pattern for this grid cell using various parameters (see Figure 1). Given the pattern and the u, v coordinates we can compute the corresponding color. Patterns do not necessarily have to be aligned on the grid: By transforming the u, v coordinates within a cell it is possible to translate, scale and rotate patterns. All these computations can be done on graphics hardware using a *fragment program* which computes the final color of a pixel from texture coordinates, texture data and other per-pixel input information.

Depending on the application, numerous parameters will influence the choice and the positioning of a pattern: Material type, probability distribution, time, distance from a point, ... As a result the (very large) resulting texture never has to be explicitly generated and is only evaluated at runtime for rendered pixels. Moreover, there are no constraints on the mesh as all computations are done in texture space by the fragment program.

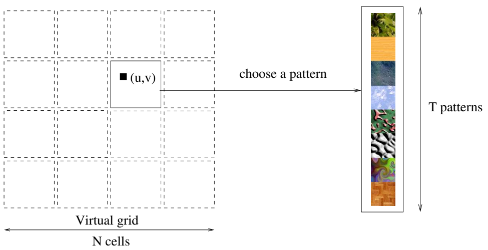


Figure 1: Pattern based procedural textures: For the cell containing the u, v coordinates a pattern is chosen on the fly.

The features required to create a wide variety of pattern based textures can be classified into three categories:

- *Choice and positioning of patterns* (section 3.2): The choice of patterns should be done either explicitly or procedurally, without showing periodicity. A local control on the probability distribution of patterns is required in order to create textures with a rich aspect from a limited number of patterns. Another important feature is the ability to translate, rotate or scale patterns within their virtual grid cell in order to cancel out the regularity introduced by the grid.
- *Transitions* (section 3.3): Tiling of square patterns (i.e. *tiles*) is typically used in video games to texture large areas with different types of material (grass, sand, ...). As some areas have different material, it requires a transition at pixel level in order to not see the border of the grid cells.
- *Animation* (section 3.4): Patterns can be used for the creation of animated textures: Each pattern is replaced by a set of patterns representing its animation. As there can be a large number of patterns displayed at the same time, this requires the ability of animating multiple patterns in an asynchronous way.

In order to not restrict the user to very specific texture operations, we propose a framework based on a set of basic *blocks*. Each block is a small independent algorithm that is in charge of a specific functionality. Designing a procedural pattern based texture consists of plugging these blocks together in order to determine a color from the u, v texture coordinates.

Using these blocks, we will show in section 4 how to create fragment programs able to compute procedural aperiodic grids of tiles (widely used in video games to create landscape textures) and random positioning of patterns on a texture (sparse convolution), while allowing a strong user control.

The following sections detail all our blocks. Next to each basic block name a diagram illustrates the block inputs and outputs. We also introduce some utility blocks composed of several basic blocks.

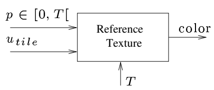
3.1 Inputs and outputs of blocks

In the following we explain our model in the 1D case. We will describe in section 3.5 how to deal with 2D or 3D.

To better differentiate between different types of textures we call textures that store color values *textures* and textures that encode of spatial distribution of parameters *maps*. For clarity we assume that a map of size N is a function from $[0, N[$ to $[0, M[$ where M is the maximum value that can be stored in the map (implementation details are given in section 3.6). The choice of M is determined by the purpose of the map. We also assume that the domain wraps, i.e. for a map of size N, $map(x+N) = map(x)$. We call a texture or map *explicit* if it stored in memory and *virtual* (or *procedural*) if it is evaluated on the fly.

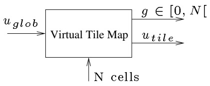
Our blocks can be easily combined: Any input parameter can use the output of another block and any parameter can be replaced by a virtual or an explicit map if extra controls are needed.

3.2 Choice and positioning of patterns



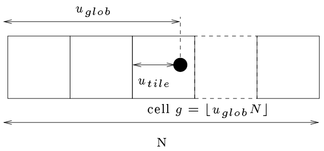
Reference Texture

The goal of this block is to simplify the management of patterns. Instead of having each pattern in one separate texture, the user provides a single texture where he packed the T patterns he wants to use. We assume that all the patterns have the same size (in pixels). Given a pattern index p and a texture coordinate $u_{tile} \in [0, 1]$ relative to the corner of the pattern this block retrieves the color of the pattern p pixel at u_{tile}.

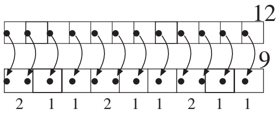


Virtual Tile Map (see Figure 2.1)

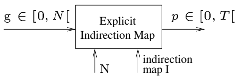
A *virtual tile map* of size N is a virtual grid of N cells (N^2 cells in 2D) that covers the entire texture space. u_{glob} is the texture coordinate that directly comes from the graphics hardware pipeline. This block computes in which cell of the virtual grid the u_{glob} coordinate lies. It also computes the coordinate u_{tile} of the pixel that is under the global u_{glob} coordinate within the cell (its *relative* coordinate). If the virtual tile map has a size of N, then the pixel is in the cell $g = \lfloor u_{glob} N \rfloor$ and its relative coordinate is $u_{tile} = frac(u_{glob} N)$ (where $frac(x)$ extracts the fractional part of x).



2.1: The Virtual Tile Map block computes the position of u_{glob} within the virtual grid of size N.



2.2: With n=12 and n'=9 aliasing occurs yielding bias in the distribution.



Explicit Indirection Map

This block allows the user to explicitly choose which pattern should appear in each cell of the virtual tile map. Let I be a user defined (i.e. explicit) *indirection map* of size N: a value in I encodes the pattern number p to be used in the corresponding cell. The pattern chosen at the grid location g is thus simply $I(g)$.

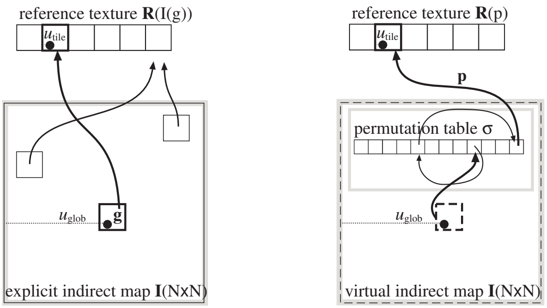
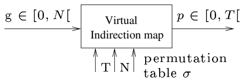


Figure 3: *Left: Explicit Indirection Map* indexing the *reference patterns*. *Right: Virtual Indirection Map* the values are computed by hashing the tile index using the *permutation* σ.



Virtual Indirection Map

In order to create large textures from patterns without showing repetitiveness, we have to be able to choose patterns in an aperiodic manner. This is done using this block: An aperiodic random number p is generated using grid index g as a seed value. This number is used as a pattern index. Therefore, a pattern is associated with each cell of the virtual tile map without showing periodicity in the pattern choice (see Figure 10).

The size of the virtual tile map N and the maximum value T of p are given. To determine the value of p (which is constant within a cell) we rely on a pseudo-random number obtained from a hashing function σ of the tile location g. We provide a table σ of size T which gives a permutation of the indices between 0 and $T-1$. The new index of i is given by $\sigma(i)$.

To account for numbers greater than T without showing periodicity, we hash g in the same way as 2D coordinates x, y are hashed by using $\sigma(x + \sigma(y))$ in [Perlin 1985] to avoid correlation: we evaluate the series $s_0 = \sigma(g)$, $s_i = \sigma(\frac{g}{T^i} + s_{i-1})$ up to the rank such that $T^{i+1} \geq N$. In practice we can unroll 2 or 3 steps, using for instance $\sigma\left(\frac{g}{T^2} + \sigma\left(\frac{g}{T} + \sigma(g)\right)\right)$. This provides an aperiodic tiling of patterns 0 to $T-1$ within the virtual grid of size N.

Virtual Random Map

The random number generated by the Virtual Indirection Map block can also be used to control any parameter like translation or scaling.

For floating point parameters (e.g. rotation, scaling, ...), we consider the result of σ as a random number in $[0, 1[$ by computing $\frac{r}{T}$. Note that this provides quantized values since there are only T entries in σ. If a better resolution is required, then a larger σ map has to be used which size replaces T in the formula.

If such a floating point value is used to generate an integer parameter (e.g. an index within another table of size T') it is better to use a permutation table σ adapted to the T' range: Otherwise aliasing could occur because the floating point number would have a quantization of $\frac{1}{T}$ and would be used as an index in $[0, T'[$. It would result in non-uniform random distributions. (see Figure 2.2). Note that if T is a multiple of T' then no problem occurs.

Uncorrelated random parameters can be obtained by using different permutation tables σ or by adding a large offset to g (since distant locations in the aperiodic tiling are uncorrelated). Note that since the hardware achieves vector operations (up to size 4), several permutations can be computed in parallel using a vectorial σ map.

Tile Transform

This block allows to scale, rotate or translate a pattern within its virtual tile map cell. It is not designed to create continuous textures but to create textures showing objects on a transparent background. It is used to approximate a Poisson distribution of patterns in a virtual texture (see Figure 14). The transformation of each pattern can be either explicit or procedural.

The block can apply scaling, translation and rotation (in 2D and 3D) on the coordinates u_{tile}. If the transformed coordinates u'_{tile} no longer lies inside the pattern the fragment is discarded (no pixel is drawn on screen).

Given a translation d and a scaling factor s for a cell, the new coordinates could be evaluated as $u'_{tile} = (\frac{u_{tile}}{s} + d)$. But proceeding this way, the pattern could go partly outside the cell and be clipped. There are two ways of avoiding this (see Figure 4):

- Shrinking the pattern to a given scale s and allowing only transformations for which it remains strictly inside the cell: $u'_{tile} = \frac{1}{s}(u_{tile} + (1-s)d)$.
- Managing the overflow through *multipass*, by combining the overflows from the neighborhood. The idea is to render the two parts of the pattern (four parts in 2D) by rendering the grid two times (four times in 2D). The first pass is done as usual. The second pass shifts the virtual grid one cell to the right and uses $d-1$ as translation (see Figure 5). The final color is then the sum of the contributions of each pass. (a blend equation can also be used to make possibly colliding patterns hiding each others).

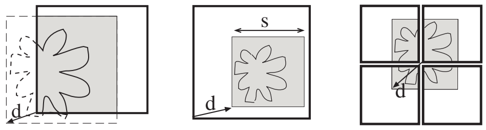


Figure 4: *Left:* A translation d can make the pattern clipped. *Middle:* Scaling s and constrained translation. *Right:* Overflow management of neighboring cells.

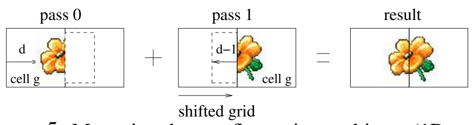


Figure 5: Managing the overflow using multipass (1D case)

Explicit Positioning

The *explicit positioning* of patterns relies on a user defined *positioning map* to place patterns at arbitrary positions in the virtual texture. This should not be confused with simple indirection maps where patterns are grid-aligned. In the case of explicit positioning patterns can be at *arbitrary positions*.

The idea is to encode a pattern position into an indirection grid and to add a local translation to the pattern. We rely on an explicit indirection map to access the reference pattern texture. The explicit position map encodes in each grid:
- the pattern index
- the offset to be applied to the pattern

If no pattern is present a special index is used (-1). If a pixel of the virtual texture does not lie in a pattern it is discarded, resulting in a transparent pixel (note that pixel in a pattern can also be transparent).

To position the top left corner of a pattern at given u, v coordinates, we need to update four cells of the explicit positioning map (see Figure 6). Allowing n patterns to be present in a same cell would require the use if n grids. Once the pattern position is set, it can still be scaled, translated and rotated by plugging the tile transform block after the explicit positioning block.

Note that explicit positioning yields the concept of *texture sprites* [Neyret et al. 2002] — i.e. sprites inside the texture space — since it allows to instantiate and move patterns within a texture: The positioning map can be modified dynamically by the user (see Figure 21 right), or the patterns position can be modified at each time step by software. *Texture sprites* are an important feature because numerous applications (3D painting on surfaces, video games, ...) need to add local details to a surface. In video games the classical approach is to rely on small transparent textured quads that are drawn on top of the geometry. Such geometrical elements are often called *decals*. They are also used to simulate various dynamic effects (footprints, signs, bullet impacts, ...) that actually are 2D sprites on surfaces. Decals require a precise positioning in space in order to appear *on* the surfaces. They introduce additional geometry for non geometrical reasons. Using our explicit positioning block no additional geometry is needed as patterns will be added in a layer on top of the uniform surface texture.

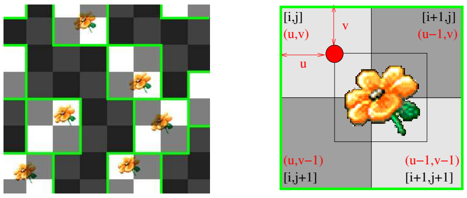


Figure 6: Four cells need to be updated to arbitrary position a pattern.

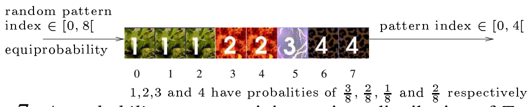
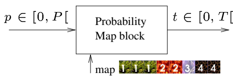
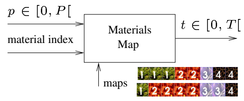


Figure 7: A *probability map* containing a given distribution of T patterns is indexed by a random pattern number in $[0, P[$.



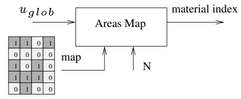
Probability Map

Using a virtual indirection map all the patterns have the same probability of occurrence since our random generator σ generates numbers with uniform probability. The purpose of this block is to control the probability of occurrence of each pattern. The idea is to replace the reference texture by another one in which the patterns are virtually duplicated in respect to the desired proportion. This is implemented using an explicit indirection map: A probability map simply contains the indices of the patterns to be used (see Figure 7). The size P of this map is arbitrary and has to be tuned in order to balance the memory size against the quantization of probabilities (the smallest not null probability is $\frac{1}{P}$). As mentioned above, such a probability map can be plugged in at any place.



Materials Map

We can consider a probability map as a *material* (e.g. 'grass with some flowers'). We want to allow the user to manipulate these materials like he does with patterns: Considering a set of materials (equivalent to the set of T patterns), a *material map* allows the user to access a material using an index.



Areas Map

An *areas map* specifies which material to use in different areas. Areas maps can be either explicit or virtual. An area map is different to an indirection map in that its resolution can be quite rougher: each cells covers an area of several tiles. If the areas map has a size of A, then the size N of the virtual tile map should be a multiple of A.

Dithered Areas Map

Areas maps are likely to be at rough resolution. If one wants to avoid sharp transitions and steps, we have to define how to interpolate the materials between the cells of the map.

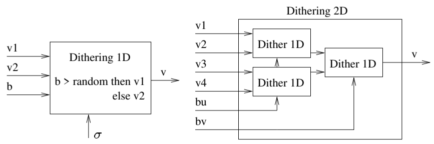


Figure 8: The dithering operators.

Which kind of interpolation to use ?
- Linear interpolation is not valid in this case as we are working with pattern indices.
- Blending of the resulting patterns can be done instead, but this often yields poor results.

For discontinuous parameters such as a pattern index, a convenient way is to rely on a dithering operator. The 1D Dithering operator (see Figure 8) takes two values and an interpolation parameter b. It relies on probabilities to achieve dithering: It computes a random number using σ and compares it to b. If b is smaller than the random number, the first value is taken, the second otherwise. Using this technique we extend the previous Areas Map block to obtain a *Dithered Areas Map*.

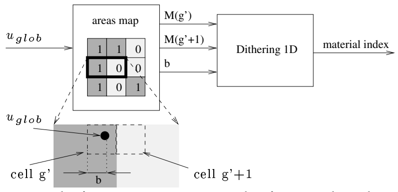


Figure 9: Low resolution areas map can be interpolated using a dithering operator which selects one of the closest values with a probability depending on the location.

To control the interpolation between the adjacent materials we compute we compute b as the barycentric coordinate of the pixel location within the neighborhood g',g'+1 (see Figure 9). Then we use the dithering operator in order to choose between the neighboring values (two in 1D, four in 2D).

Figure 10d shows the intermediate probability distributions produced by the dithering operator in the transition areas.

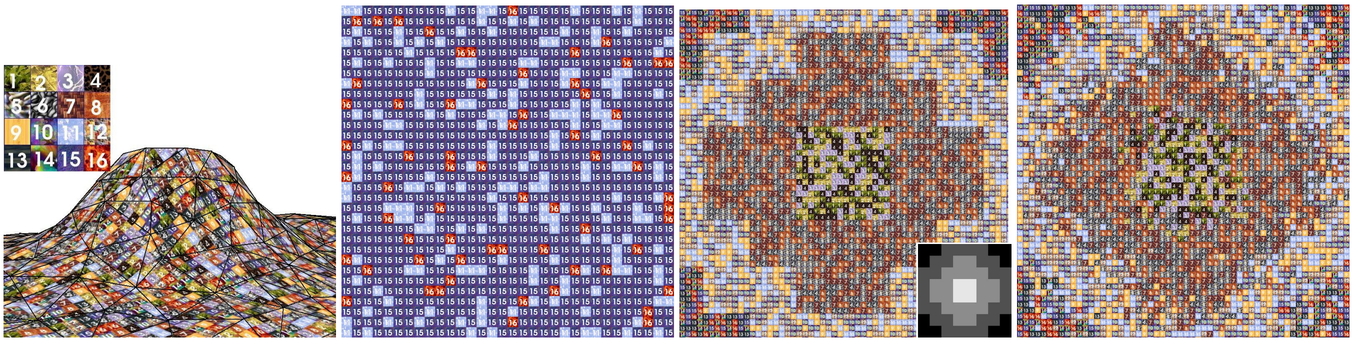
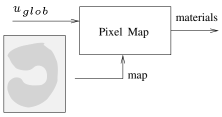


Figure 10: *From left to right*: *a:* Aperiodic tiling of 16 patterns on a terrain. Since everything is done in texture space, it is independent of the mesh. *b:* Non-uniform distribution using a probability map. *c:* Non-stationary distribution using an 8×8 areas map (shown on bottom right). *d:* Same with dithering interpolation of the areas map (the resulting virtual texture is 4096×4096, and could be far greater).

3.3 Transitions

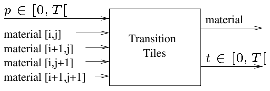
In order to create a continuous texture when using a random tiling, all tiles should be 'compatible' (left-right and bottom-up edges of all patterns should correspond to each other). But this supposes that the texture is quite homogeneous, while landscapes are generally composed of several different areas of homogeneous material (e.g. forest, lawn, beach, lake): The constraint of compatible tile edges makes sense within a area, but not on the area boundary. We introduce the notion of *material family* corresponding to a set of compatible tiles which can form a homogeneous material area. For landscapes we consider a set of material families. The purpose of the following blocks is to deal with the transition at the interface between two families.

Two different blocks can be used: The first one relies on a very high resolution *pixel map* which texels indicate the material family to use (it is a *mask* with a sharp transition). The second is based on extra tiles called *transition tiles* specifying the transition between given material families. It is inspired from a technique used in video games.



Pixel Map

These maps are used as masks to delimit precisely the shape of the areas corresponding to a material family. They are of very high resolution, possibly the same than the virtual texture. However, they compress strongly using the indirection texture mechanism of [Kraus and Ertl 2002; Cool 2002]: Only the boundary of the areas needs to be stored. Pixels Maps differ to Area Maps in that they apply sharp transitions at the pixel level while Area Maps apply transitions between materials at the tile level or rougher.



Transition Tiles

To use this mechanism, the user provides a set of tiles that explicitly describe the transition between material families (see Figure 11). These new tiles are treated as usual: A new material is defined to access these transition tiles. The transition tiles block checks if a transition is needed and then chooses the right transition tile.

To achieve this task, we rely on an idea used in games: The transition tile is chosen by looking at the material of the neighboring cells. A transition texture dealing with m families of materials therefore contains m^4 transition tiles in 2D (including the m plain tiles). (there are 16 combinations for a 2×2 neighborhood in 2D, including the 2 ones corresponding to 'material family one only' and 'material family two only'). The transition tile index is built by combining the material families indices on a logical basis (see Figure 11). If plain tiles are selected (the four neighbors are equal) then no transition is needed and the block is a pass-through.

In practice it is often sufficient to describe the border of the material family with a transparent background and to use it as a transition with all other material families. If the pixel is on the background it takes the value of the material of the other family at this location. Figure 15a top right shows an example of transition tiles that describes the border of a material family with a black background.

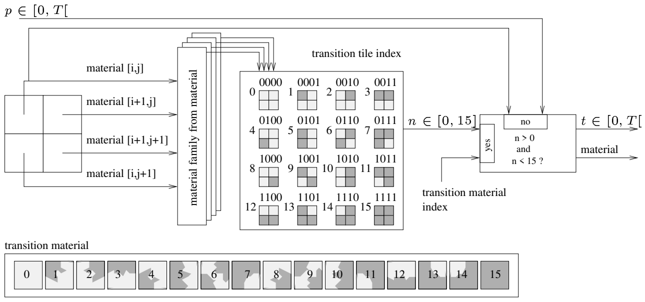


Figure 11: The transition tile index is computed from the neighborhood. If all neighbors have the same material family, the block copies its inputs on its outputs (pass-through).

3.4 Animation

Animation sequence (see Figure 12 and 21)

We can deal with patterns whose appearance depends on extra parameters (e.g. time, view direction, distance to a target...). This is done by replacing each pattern by a texture which encodes the sequence of appearances of the pattern when the parameter varies within a quantized range. The first pattern of the sequence is considered as the pattern at rest (e.g. motionless object, or background color).

The same idea as probability maps can be used to change the original time line of the animation: In particular, it is possible to change the time during which a pattern of the animation is displayed.

In order to control the animation sequence provided by the user, we introduce two parameters: The *phase* parameter ϕ adds an offset to the animation sequence time line and the Z parameter represents the amount of rest states to add at the beginning of the animation cycle (see Figure 12). Both parameters can be defined using either an explicit map or a random number.

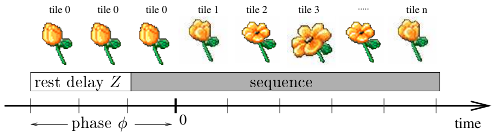
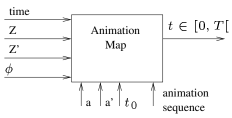


Figure 12: Animation sequence with $\phi = 3$ and $Z = 2$.



Animation Map

When using animation sequences, the user would often prefer to avoid the synchronization of the animation of all patterns. Moreover he probably wants to control the triggering of animation cycles instead of having everything animated everywhere simultaneously. To obtain asynchronous animations a different ϕ parameter is used at each location. This is done by using a random map. The Z parameter is also varying at each location to obtain different triggering times. To ease the global control we multiply Z by a global parameter a that can be tuned by the program on the fly to control how often the animated sequences are triggered (density of animation).

The pattern number to be used at time t is thus $p = MAX(\ 0,\ (t+\phi) mod(n+aZ) - aZ\ )$ where *mod* is the modulo operator and n is the length of the sequence without the leading aZ blanks.

Temporal Transitions: At this stage we have an animation with a constant pace. For an interesting animation the user probably wants to animate the various parameters (a,Z...) over the running time of the program. The main problem is to achieve this without introducing discontinuities (popping) when switching from a cycling animation to another in a given place and time. I.e. we have to define *temporal transitions*. The point is that we cannot have state variables[2] associated with patterns so we cannot store the latest step of an animation used at a given location: We have to avoid discontinuities by choosing at any time between the old and the new animation cycle (see Figure 13).

Let a' and Z' be the old parameters and a and Z the new ones after the transition at time t_0. The pattern to be used according to the old cycling is $p'(t) = MAX(\ 0,\ (t+\phi) mod(n+a'Z') - a'Z'\ )$ while the new one is $p(t) = MAX(\ 0,\ (t+\phi) mod(n+aZ) - aZ\ )$. Due to the worst case the transition lies between t_0 and $t_0 + 2n$, during which we have to choose, at each time step and for each location, either p or p', or possibly nothing. Let t_f be the time when the old cycling finishes: $t_f = t_0 + n - p'(t_0)$. If $p'(t_0) = 0$ we take $t_f = t_0 - 1$ in order to stop even before the old cycle starts. As long as $t \le t_f$, we stick to the old cycling. If $p(t_f) = 0$, we can switch directly to the new cycling as soon as $t > t_f$. Otherwise a collision occurs between the two cycles so we cannot switch immediately to the new one. In such a case we display nothing (i.e. rest state) up to the theoretical end of the newly started cycling, i.e. up to $t_{end} = t_f + n - p(t_f)$.

Each time the user wants to change the Z map (the pattern of animation) or a (the global rate of animation), he has to keep the old values a' and Z' and the transition time t_0, and then he should provide these together with the new a and Z when drawing the virtual texture. Since dealing with double collisions would require extra tests, the user should keep a given set of parameters during at least $2n$ time steps.

[2]Such a feature could be implemented with off-screen rendering, which is not in the spirit of procedural textures. Moreover it would be resource consuming due to the cost of the extra rendering pass and to the potentially large explicit map that would be generated.

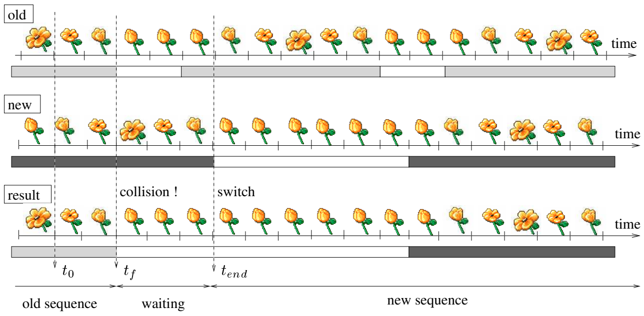


Figure 13: Transition between the old and the new cycling when requested while avoiding popping.

3.5 From 1D to 2D

For clarity we explained our representation in 1D. In the 2D case some tables corresponding to lists (e.g. probability maps) would remain 1D while maps and grids would be 2D. Yet, we chose to encode every structures in 2D. Apart from symmetry reasons (easing the plugging ability as well as the extension to higher dimensions), this optimizes the dynamics of the texture indices. In fact this also optimizes the computation cost since the graphics hardware allows for vectorial operations (which requires that identical operations be done on each component).

For instance, the reference texture R is a list of T patterns. We encode it as a $T_x \times T_y$ 2D packing of the patterns. For convenience, we then define a pattern by its coordinates (p_x, p_y) in R rather than its rank p in the list. One can think of these 2D coordinates as a *vectorial pattern index* $\overrightarrow{p}$. We do the same for each kind of table: for instance, indirect maps I encode a pattern index. Since this index is a vector, this implies that I is a function from $[0, N_x[\times[0, N_y[$ to $[0, T_x[\times[0, T_y[$ (where $N_x \times N_y$ is the size of the virtual tile map and $T_x \times T_y$ the number of patterns). Similarly, permutation tables σ used for virtual maps are now vectorial: the first component σ_x is a permutation of the horizontal indices (between 0 and $T_x - 1$) while the second (σ_y) is a permutation of the vertical indices (between 0 and $T_y - 1$), so that $\overrightarrow{\sigma}(\overrightarrow{p}) = (\sigma_x(\overrightarrow{p}), \sigma_y(\overrightarrow{p}))$.

Let us extend scalar 1D functions such as $+, /, \lfloor\rfloor, frac$ to vectors by considering that they apply to each component separately. We use the following notation: $\overrightarrow{a}\,\overrightarrow{b}$ for the component by component product of vectors $\overrightarrow{a}$ and $\overrightarrow{b}$; $\overrightarrow{u} = (u, v)$; $\overrightarrow{n} = (n_x, n_y)$ and $\overrightarrow{N} = (N_x, N_y)$. With these notations the entire formalism explained in 1D now applies to 2D. The extension to higher dimensions is similar.

Since the formulas are identical independent of the dimension we will no longer use arrows on vectors: a or uN rely to scalars for 1D textures and to vectors for 2D textures. a_x and a_y denote the components of a, and $a + (1,1)$ means $(a_x + 1, a_y + 1)$.

3.6 Encoding maps as textures

For simplicity we described a map of size N as a function from $[0, N[$ to $[0, M[$ where M is the maximum value that can be stored in the map. Actually all maps are encoded in textures and are functions from $[0,1]$ to $[0,1]$. The conversion is straightforward: if a map of size N and maximum value M is accessed by an index i, then $\frac{i}{N}$ should be used to access the corresponding texture. The integer corresponding to the floating point value $r \in [0,1]$ read in the texture is simply $\lfloor rM \rfloor$.

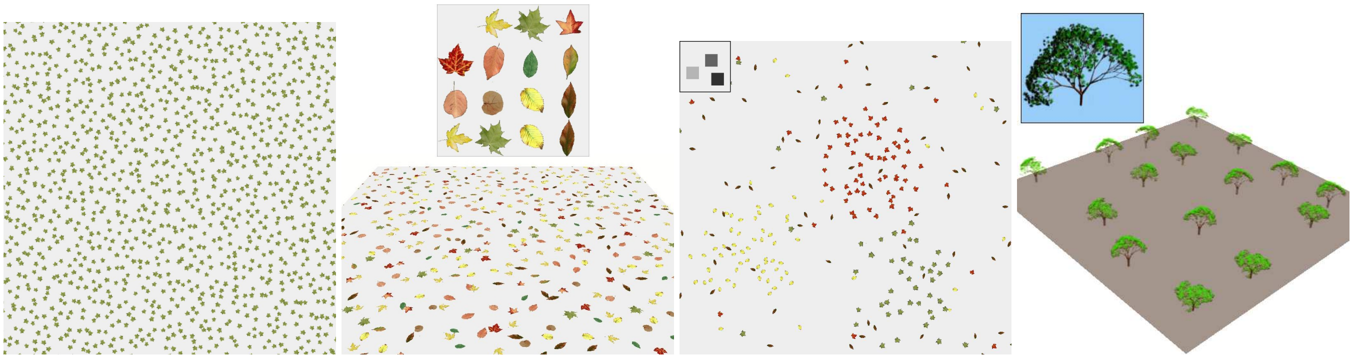


Figure 14: *From left to right*: *a:* Sparse convolution of a leaf pattern (plus random rotations). *b:* Sparse convolution of random leaf patterns using a probability texture. *c:* Non-stationary sparse convolution using a 8×8 probability map (shown on top left) with dithering interpolation. *d:* Sparse convolution of volumetric textures (the tree pattern is 256^3. The terrain is rendered using 256 slices, i.e. 256 quads).

4 Combining blocks: Cases studies

In the previous section we have defined a set a pluggable blocks. Using the CG fragment programs language [Nvidia 2002a] these blocks correspond to CG functions. We illustrate now how to combine them by implementing two major texture types: Aperiodic tiling and sparse convolution in 2D. For both we propose a simple version (respectively random tiling and Poisson distribution) and an extended version allowing various user controls. We also describe how to create animated textures from animated patterns.

Basic aperiodic tiling: (see Figure 3 right, 10a)
The user provides a reference texture which contains $T \times T$ packed patterns. He wants to tile them along a virtual grid of size $N \times N$ aperiodically with a uniform distribution. The needed blocks are:
- one *Virtual Tile Map* block used to compute the cell coordinate g and the relative coordinate u_{tile} from the u_{glob} coordinate.
- one *Virtual Indirection Map* block used to compute a random tile index p from the cell coordinate g.
- one *Reference Texture* block used to compute the final color from the tile index and the relative coordinate u_{tile}.

The diagram of this fragment program is given in Figure 16. This scheme is also figured on Figure 3 right.

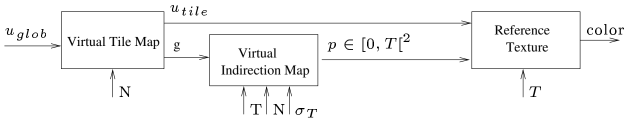


Figure 16: Basic aperiodic tiling

The fragment program can be written from the diagram:

```
void fragment_program(float2 u_glob,out:float4 color) {
   float2 g,u_tile,p;
   virtual_tile_map(u_glob,out:g,out:u_tile);
   virtual_indirection_map(T,N,sigma,g,out:p);
   reference_texture(T,p,u_tile,out:color);
}
```

Extended aperiodic tiling: (see Figure 15)
The user defines 2 material families and 2^4 transition tiles. He also defines materials (probability maps) inside each material family (see Figure 15a left). He provides a rough areas map M (to be interpolated) in which he painted the areas of each material (see Figure 15a bottom-right).

The purpose of this fragment program is to tile randomly grass tiles within dark areas of the areas map and sand tiles within light areas, using appropriate transition tiles at the boundary between the 2 areas (see Figure 15 b and c). To achieve this the idea is to check at first whether a transition is needed or not. This is done by the *Transition Tiles* block. Its inputs are a random index computed by the *Virtual Indirection Map* block and the materials of the cell g and its three neighbors. These materials are computed by four copies of a *Dithered Areas Map*. If a transition is needed, the *Transition Tiles* block returns the transition material index and the transition tile index. If not, the material of cell g is returned together with the random number generated by the *Virtual Indirection Map* block. The final color is computed by a *Reference Texture* block from the tile index generated by the *Materials Map* block.

The diagram of this fragment program is given in Figure 17.

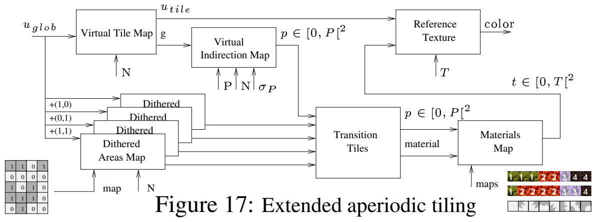


Figure 17: Extended aperiodic tiling

Basic sparse convolution: (see Figure 14a)
The reference texture contains one pattern ($T = 1$). The user wants to distribute the pattern on the surface according to a Poisson disk distribution of radius p. The Poisson disk distribution is approximated by choosing a random location in each cell of a grid of size $N \times N$ with $N = \frac{1}{p}$ (as classically done in the scope of stochastic sampling [Cook 1985]). The diagram of this fragment program is given in Figure 18.

The *Virtual Random Map* block is used for computing a vector of floating point random numbers (using vectorial arithmetic four permutations can be computed simultaneously). These numbers are used to specify a translation (two numbers) a scaling and a rotation (one number each). In practice we use a permutation σ with a size of 256. It allows a resolution of $\frac{1}{256}$ for the generated random numbers. The *Tile Transform* block transforms the tile within its cell g. It returns updated relative coordinates u'_{tile} or discrads the fragment.

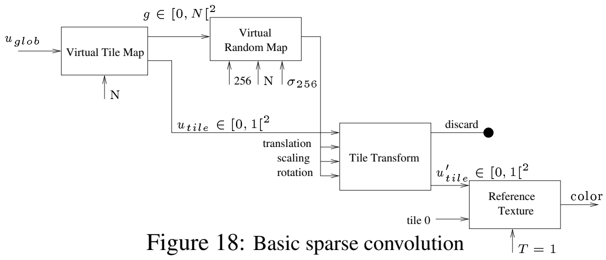


Figure 18: Basic sparse convolution

Extended sparse convolution: (see Figure 14c)
The user wants to control the density of the distribution along the virtual texture. For this he defines some materials corresponding to typical proportions of the various patterns and empty space (i.e. blank pattern). He also paints an areas map M at rough resolution to

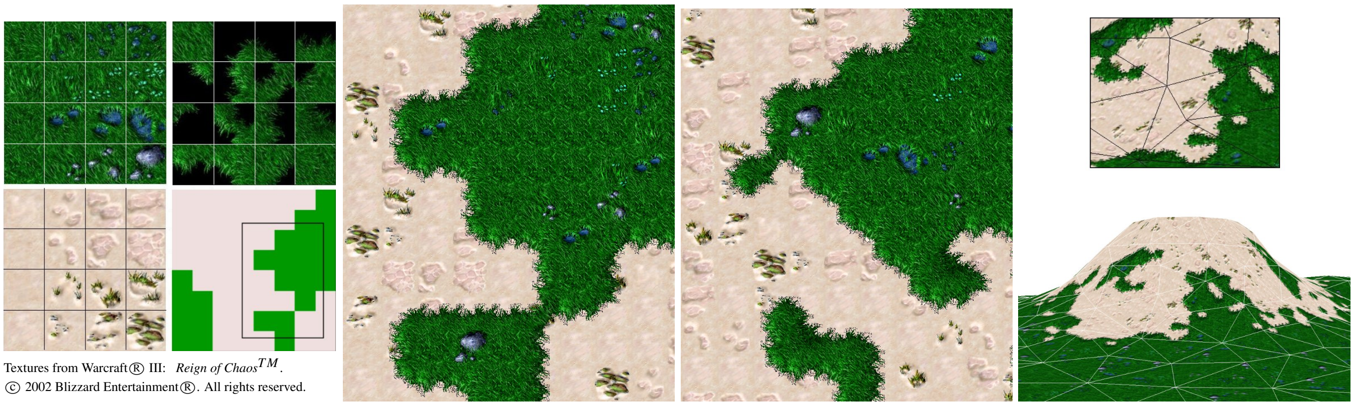


Figure 15: *From left to right*: *a:* Regular and transition patterns (extracted from the game Warcraft 3) and the 8×8 probability map. *b:* Zoom on a detail of the virtual texture (the geometry is a single quad). *c:* Dithering interpolation of the map. *d:* Mapping of the virtual texture (4096×4096) on a terrain.

specify the areas of various materials. The corresponding algorithm is built upon the basic sparse convolution. The interpolation of M produces a continuous interpolation of the probabilities.

The added blocks are:

- a *Virtual Indirection Map* block used to produce a random index in each cell g.
- one *Dithered Areas Map* block used to choose the local material.

The diagram of this fragment program is given in Figure 19.

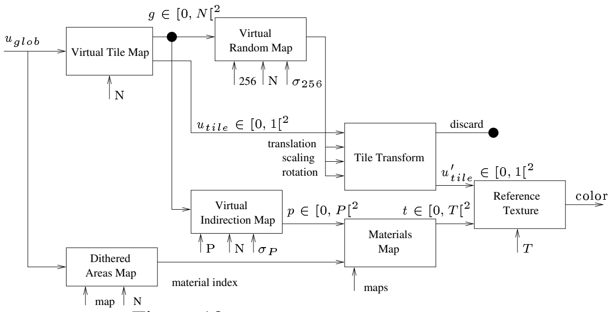


Figure 19: Extended sparse convolution

Animation:

Two major kinds of animation can be handled:

- Long range explicit displacement of a pattern (see Figure 21d).
- Stationary animation of patterns (see Figure 21a-c).

• The first case corresponds to the dynamic use of the explicit positioning block. As explained in section 3.2, no extra geometry is required to display the moving *texture sprites* (contrary to games) and very few parameters need to be sent by the program to update the virtual texture (contrary to [Neyret et al. 2002]). The motion is obtained simply by updating the concerned cells in the low-resolution positioning map (four cells for a single pattern, which corresponds to very little data).

• The second case corresponds to the idea of *textural motion*: Any parameters such as scale, rotation or location can be explicitly defined as a function of time. The Animation Sequences defined in section 3.4 can also be used. We focus here on *time cyclical* pattern animation, relying on Animation Maps.

The complete diagram of the fragment program is given in Figure 20. The parameters *time*, Z, Z', a, a' and t_0 are handled by the software (they can be animated, see section 3.4).

5 Texture filtering

Hardware texture filtering does not work well when using indirections. Therefore, all techniques relying on this functionality suffer

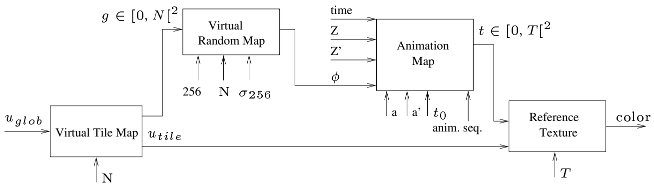


Figure 20: Aperiodic animation of a grid of tiles

from filtering issues because indirections introduce discontinuities in the u, v.

The problem comes from the wrong neighborhood information. If we use standard filtering, the color used by the hardware is not taken from the right location: It uses the pixel which is the neighbor in the reference texture instead of the pixel which is the neighbor in the virtual texture. This implies that both linear interpolation and MIP-mapping are incorrect.

The problem of filtering with indirection was previously encountered in [Kraus and Ertl 2002]. The authors propose to duplicate the boundary of tiles to ensure correct linear interpolation. Nevertheless some aliasing artifacts are still visible. MIP-mapping is not addressed in this paper. Note that octree textures [Benson and Davis 2002; DeBry et al. 2002] could also be implemented using hardware indirection but here again the filtering would be an issue.

5.1 Filtering our procedural textures

In the following we call *texel* one pixel of a texture. The filtering scale depends on the number of texels that are projected on each pixel of the screen. The ideal case is when one texel is projected on one pixel so that no aliasing occur.

Note that the classical solution for aperiodic tiling in games relies on quads on which different patterns are applied (by using appropriate texture coordinates). Pixels adjacent to borders are not correctly filtered either: the only current solution is to oversample the pixels. MIP-mapping also has the same limitations when a pattern reduces to a pixel: Textural aliasing turns into geometrical aliasing as multiple quads are projected into the same pixel. This implies that our method cannot be worse than the currently available methods in terms of filtering.

• **less than one texel per pixel:** To correctly handle the interpolation between texels a first approach would be to rely one the *ddx* and *ddy* operators (on GeForceFX) to replace the hardware interpolation in the fragment program. Nevertheless it would require four evaluations of the *entire* fragment program in order to evaluate the color of neighboring pixels.

In practice we often use tiles with compatible edges: the pixels in a margin have similar colors through patterns. Patterns prepared for random positioning have a similar property since their margin has a transparent or background color. In this case linear interpolation

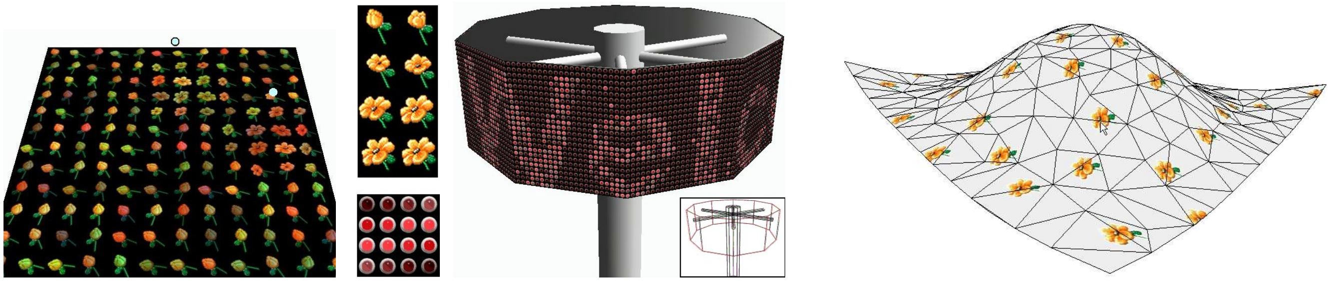


Figure 21: *From left to right*: *a:* The sequence textures are triggered by the distance to the two balls. *b:* The sequence patterns used for images *a* and *c*. *c:* A LED message board with blinking LEDs (the virtual texture is 12288×1536). *d:* Interactive positioning of patterns.

works correctly at the firsts MIP-mapping scales: A wrong texture pixel is used but it has the correct color. However even in this case there exists a scale from which the continuity margin vanishes (as mipmap levels converge to a unique pixel of average color).

We discuss in section 5.2 how an extension of the hardware could allow a general solution for interpolation with indirection.

• **several texels per pixels:** Tiles are packed in a unique texture. This implies that there exists a MIP-mapping level for which each texel corresponds to the average color of the underlying tile (we assume that all tiles have the same size which is a power of two). All MIP-mapping levels coarser than this one are wrong as they no longer correspond to the filtering of tiles: Their texels contain mixed colors from several tiles. It is thus important to prevent the hardware from using this MIP-mapping levels (using the `GL_TEXTURE_MAX_LOD` feature of `glTexParameter`).

• **several patterns per pixels:** When the viewpoint is far from the textured object, more than one pattern is projected onto each pixel. As stated above the MIP-mapped pattern texture cannot be used above the scale for which one tile is represented by a single texel t: the indirection map also has to be filtered. At this level we can forget the individual patterns and the indirection map can be considered as a regular map based on texels t. We thought of three solutions to sum the colors on the projected pixel area:

- Oversampling these texels. This makes sense if only a few are needed.
- Precalculating an explicit MIP-mapped map using these texels in the spirit of clips-maps [Tanner et al. 1998] (its resolution should be reasonable).
- In the case of areas maps, taking advantage of the law of large numbers (which is valid if numerous patterns project on a single pixel): The color tends towards the average color of each material. Thus the areas map can directly be used assuming the average materials colors have been precalculated.

5.2 Discussion on texel interpolation

To deal correctly with the interpolation we have to explicitly access and combine the texel values, bypassing the automatic `LINEAR` interpolation mode. This could have been simpler without the limitations of the current hardware: the idea would have been to add a black margin to the patterns, then to superimpose the margin of adjacent tiles (this implies that some pixels would have to sum two or four contributions). This should be done at any MIP-mapping level.

The hardware limitation disqualifying this solution is that the notion of *sub-texture* does not exist: The `BORDER_COLOR` feature allows to simulate a black border without using extra memory but it only applies to the border of the global texture in which the patterns are packed. We cannot add such a border explicitly to each tile (the texture size will no longer be a power of two) and we cannot put each tile in one texture because there is a limited number of texture usable at the same time.

Actually we think that in order to correctly handle filtering with indirections whatever the application ([Kraus and Ertl 2002; Benson and Davis 2002; DeBry et al. 2002], ours), there is a need for a new texture format that would introduce the notion of sub-textures. Using this it would be possible to describe a texture as a set of independent sub-textures. It would therefore be possible to work on each pattern independently without having the color of neighboring tiles mixed.

6 Results

We implemented all the basic blocks and the various test applications with OpenGL and CG for Nvidia NV30 relying on a software driver emulating the future graphics board [Nvidia 2002b].

A few days before the camera ready version we received from Nvidia a prototype GeForceFX board allowing us to do some performance measurements. Note that this prototype is about 50% underclocked, and that the first version of the CG compiler does not produce optimized fragment codes. Moreover we can not implement discarding of empty fragments since this feature is not recognized by the current compiler (on sparse convolutions 40% to 70% of tiles should be discard). Thus all the timings we provide should be accelerated by a factor of 2 to 16 on the final GeForceFX and CG compiler. Times are given for images with every pixels covered by texture. These measures remain constant whatever the number of tiles is. We used floats (32 bits precision) in the fragment programs. Using halves (16 bits precision) gains approximatively 50%, but the limited precision might be visible in some cases. The following table shows that the performance are 0.4 to 0.8 Giga intructions per seconds. The cost is proportional to the number of rendered pixels.

	Basic	Areas Map	Dithered Areas Map	Dithered Areas Map & Transitions
code length	56 instr.	65 instr.	117 instr.	512 instr.
tex. lookups	5	7	10	39
320x200	113 fps	73 fps	36 fps	5 fps
640x480	24.5 fps	15.5 fps	8.5 fps	1.1 fps

All the map and texture sizes are based on powers of two to avoid precision problems with divisions. This also allows to limit the number of permutation tables σ required since a table of n indices can also be used without bias for sub-multiples of this size. The patterns we used have a square shape and are of the same resolution. Concerning the filtering, we only implemented a simple solution for the MIP-mapping (locking the deepest filtering to the level where patterns are the size of a pixel), and we did nothing special for the linear interpolation through tile borders. However, the patterns we use for landscapes (e.g. grass) have a similar margin to ease the continuity and generally have the same average color. The fact that patterns share color properties avoids most of the artifacts.

The images on Figure 10 present the aperiodic tiling of 16 patterns 64×64 with increasing amount of user control. Even in the more complex case very little memory is needed to obtain the 4096×4096 virtual map. As shown in Figure 10a the mesh is totally independent of the texture features.

The images on Figure 14 illustrate sparse convolution: random location and rotation of leaves provide another kind of non repetitive texture, which can be enhanced by controlling the spatial distribution of leaves spots. On the right we used a 256^3 volumetric texture [Meyer and Neyret 1998].

The images on Figure 15 show the use of transition patterns. Two families of 16 patterns 64×64 are used for the ground and the grass, and 16 patterns are defined for the transition. Since the resulting texture is not stored and is only evaluated where it is visible, very large game fields can be specified and explored with high resolution details. Here again the map controlling the material distribution (the areas map) is interpolated, and could have been procedurally computed as well.

The images on Figure 21 correspond to the movies available on our website. The first animation (see Figure 21a) presents a sequence texture, the parameter of which is connected to the distance to a target: The flowers open when the small balls are close to them. The second and the third animations illustrate the animation maps: The animation texture contains the various states of a blinking red LED. In the first (see Figure 21c) we rely on a large animation map containing the sentence to be displayed. The scrolling motion is simply obtained by increasing the u value depending on time in software. Note that only the visible part of the 12288×1536 texture is calculated. The noise on the panel is simulated by tuning the rest time of the LEDs: It is not null in the letters, and not infinite in the background. The other animation (not illustrated in the figure) relies on temporal transitions: 3 maps are used (2 messages plus a blank panel), and the density of animation (see Section 3.4) is smoothly increased or decreased. The geometry consists of a single quad. The fourth animation (see Figure 21d) shows the interactive positioning of patterns using an explicit map: The virtual texture can be dynamically updated easily (NB: This interactive session was captured on a limited implementation on a GeForce3). Note that the long range displacement of patterns illustrated here also corresponds to a modality of animation.

7 Conclusion and future work

We proposed a way to make very high resolution texture spaces available by defining a representation and hardware based rendering algorithms able to procedurally combine patterns. In particular this allows to implement classical features such as aperiodic tilings and sparse convolutions. As we have shown we can also offer various low level to high level controls to the user, like specifying probability distributions, maps of material distribution, animation... The blocks are general enough to allow infinite combinations, each plugged block possibly offering new user controls. Our representation is particularly convenient in the case of landscapes, the visualization of which requires showing details of the foreground as well as wide, non-repetitive features of the overall scenery. Since everything is done in texture space, the mesh is totally free of constraints and can be sampled according to geometrical criteria only.

More block types could be designed for the needs of specific applications. The animation probably has the greatest potential for extensions. We implemented color and transparency textures, but also bump maps and volumetric textures. The latter two require more complicated filtering which is intricate to the shading model. This is a topic of future work. Finally, it would be interesting to adapt some ideas of the clipmaps (caching, fetching on request) to develop a model allowing infinite zooming. More generally, indirect textures open new horizons to texture programming and for alternate representations.

Programming on the NV30 board was a great pleasure. The fragment program language almost allows general programming. As we stated in the paper the notion of sub-texture would help recognizing indirect textures as a native feature. A `noise` function is documented in the CG specification but is not yet implemented. When available, it might be usable to generate aperiodic tiling (depending on what it really does !). Conversely, our index hashing based on permutation tables could be a way to implement this noise.

The properties of this board are different enough compared to previous graphics boards as to change the strategy concerning the representation choices. For instance the `discard` feature allows to abort a fragment program as soon as it can be known that the fragment is not visible. This can dramatically decrease the fillrate cost, pushing the representations based on transparent textures (billboards, volumetric textures...). Yet, having room for 1024 instructions means that the evaluation of one pixel might get costly: For each given family of applications, we will have to compare geometry-based, billboard-based, point-based and texture-based solutions to find which can give the best performance.

Acknowledgments

We wish to thank Peter Wonka, Stéphane Guy, Alexis Angelidis and Sylvain Paris for rereading this paper, and also Xavier Décoret, Laure Heigeas and Jean-Sébastien Franco who reread the submission version. Thanks are also due to *Nvidia* for providing us with a GeForceFX, and to *Blizzard* for the tile textures used in Figure 15.

References

BENSON, D., AND DAVIS, J. 2002. Octree textures. In *In SIGGRAPH'02 Conference Proceedings*, ACM Press, 785–790.

COOK, R. L. 1985. Antialiasing by Stochastic Sampling. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*. July.

COOL, M. M., 2002. Sparse Texture Storage for Graphics Accelerators. Technical Talk. `http://www.cgl.uwaterloo.ca/Projects/rendering/Talks/sparse/slides.pdf`.

DEBRY, D., GIBBS, J., PETTY, D. D., AND ROBINS, N. 2002. Painting and rendering textures on unparameterized models. In *In SIGGRAPH'02 Conference Proceedings*, ACM Press, 763–768.

EBERT, D., MUSGRAVE, K., PEACHEY, D., PERLIN, K., AND WORLEY. 1994. *Texturing and Modeling: A Procedural Approach*. Academic Press, Oct. ISBN 0-12-228760-6.

KRAUS, M., AND ERTL, T. 2002. Adaptive Texture Maps. In *Proc. SIGGRAPH/EG Graphics Hardware Workshop '02*, ACM SIGGRAPH, 7–15.

LEWIS, J.-P. 1989. Algorithms for Solid Noise Synthesis. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, J. Lane, Ed., vol. 23, 263–270.

MEYER, A., AND NEYRET, F. 1998. Interactive Volumetric Textures. In *Rendering Techniques'98, Eurographics Rendering Workshop*, Springer Wein, G. Drettakis and N. Max, Eds., Eurographics, 157–168.

NEYRET, F., AND CANI, M.-P. 1999. Pattern-Based Texturing Revisited. In *SIGGRAPH 99 Conference Proceedings*, ACM SIGGRAPH, 235–242.

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 (may), 135–149. http://www-imagis.imag.fr/Publications/2002/NHS02.

NVIDIA, 2002. CG Toolkit Reference Manual v1.5 `http://www.cgshaders.org`.

NVIDIA, 2002. NV30 emulation: beta detonator driver (version 40.41). `http://developer.nvidia.com/view.asp?IO=nv30_emulation`.

PERLIN, K. 1985. An Image Synthesizer. In *Computer Graphics (SIGGRAPH 85 Proceedings)*, B. A. Barsky, Ed., vol. 19, ACM SIGGRAPH, 287–296.

SGI OpenGL Extension Registry:
ARB_texture_compression & EXT_texture_compression_s3tc
`http://oss.sgi.com/projects/ogl-sample/registry/`.

SOLER, C., CANI, M.-P., AND ANGELIDIS, A. 2002. Hierarchical Pattern Mapping. In *Siggraph'02*, ACM SIGGRAPH, 673–680.

STAM, J. 1997. Aperiodic Texture Mapping. Tech. Rep. R046, European Research Consortium for Informatics and Mathematics (ERCIM), Jan. `http://www.ercim.org/publication/technical_reports/046-abstract.html`.

TANNER, C. C., MIGDAL, C. J., AND JONES, M. T. 1998. The Clipmap: A Virtual Mipmap. In *Proceedings of SIGGRAPH 98*, ACM SIGGRAPH, 151–158.

WORLEY, S. P. 1996. A Cellular Texturing Basis Function. In *SIGGRAPH 96 Conference Proceedings*, Addison Wesley, ACM SIGGRAPH, 291–294.

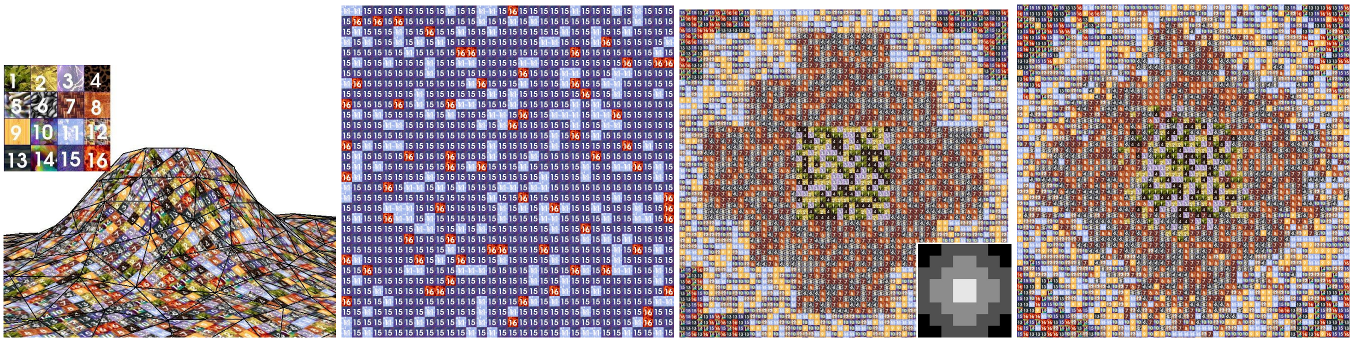


Figure 10: *From left to right*: *a:* Aperiodic tiling of 16 patterns on a terrain. Since everything is done in texture space, it is independent of the mesh. *b:* Non-uniform distribution using a probability map. *c:* Non-stationary distribution using an 8×8 areas map (shown on bottom right). *d:* Same with dithering interpolation of the areas map (the resulting virtual texture is 4096×4096, and could be far greater).

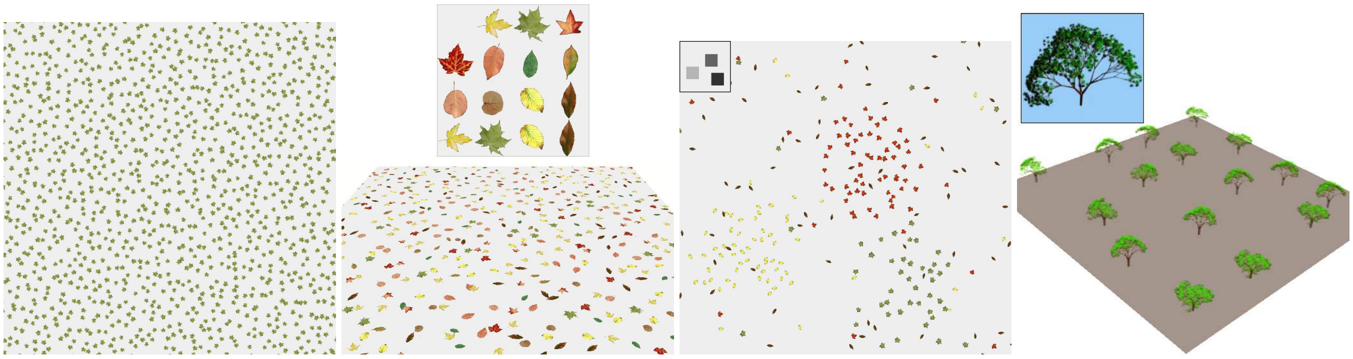


Figure 14: *From left to right*: *a:* Sparse convolution of a leaf pattern (plus random rotations). *b:* Sparse convolution of random leaf patterns using a probability texture. *c:* Non-stationary sparse convolution using a 8×8 probability map (shown on top left) with dithering interpolation. *d:* Sparse convolution of volumetric textures (the tree pattern is 256^3. The terrain is rendered using 256 slices, i.e. 256 quads).

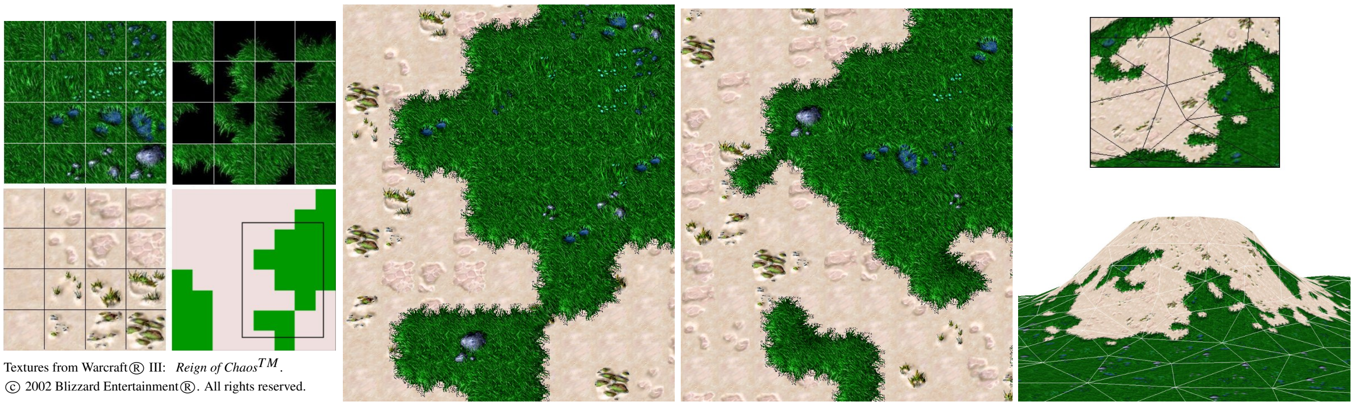


Figure 15: *From left to right*: *a:* Regular and transition patterns (extracted from the game Warcraft 3) and the 8×8 probability map. *b:* Zoom on a detail of the virtual texture (the geometry is a single quad). *c:* Dithering interpolation of the map. *d:* Mapping of the virtual texture (4096×4096) on a terrain.

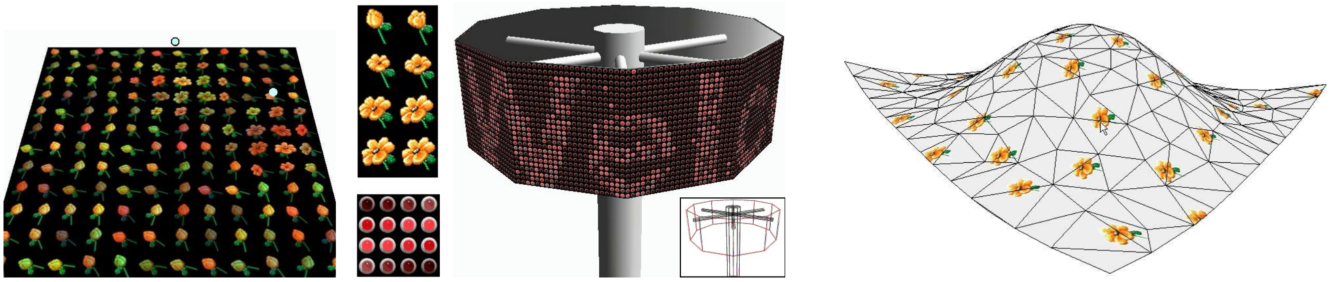


Figure 21: *From left to right*: *a:* The sequence textures are triggered by the distance to the two balls. *b:* The sequence patterns used for images *a* and *c*. *c:* A LED message board with blinking LEDs (the virtual texture is 12288×1536). *d:* Interactive positioning of patterns.