INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Unified Texture Management for Arbitrary Meshes

Sylvain Lefebvre
EVASION/GRAVIR

Jérome Darbon
ENST, LRDE/EPITA

Fabrice Neyret
EVASION/GRAVIR

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

N° 5210

May 2004

Thème COG

Rapport de recherche

N 0249-6399 ISRN INRIA/RR--5210--FR+ENG

Unified Texture Management for Arbitrary Meshes

Sylvain Lefebvre EVASION/GRAVIR
Jérome Darbon ENST, LRDE/EPITA
Fabrice Neyret EVASION/GRAVIR

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

Thème COG — Systèmes cognitifs
Projets EVASION

Rapport de recherche n° 5210 — May 2004 —20 pages

Abstract:
Video games and simulators commonly use very detailed textures, whose cumulative size is often larger than the GPU memory. Textures may be loaded progressively, but dynamically loading and transferring this large amount of data in GPU memory results in loading delays and poor performance. Therefore, managing texture memory has become an important issue. While this problem has been (partly) addressed early for the specific case of terrain rendering, there is no generic texture management system for arbitrary meshes.

We propose such a system, implemented on today's GPUs, which unifies classical solutions aimed at reducing memory transfer: progressive loading, texture compression, and caching strategies. For this, we introduce a new algorithm – running on GPU – to solve the major difficulty of detecting which parts of the texture are required for rendering.

Our system is based on three components manipulating a tile pool which stores texture data in GPU memory. First, the Texture Load Map determines at every frame the appropriate list of texture tiles (i.e. location and MIP-map level) to render from the current viewpoint. Second, the Texture Cache manages the tile pool. Finally, the Texture Producer loads and decodes required texture tiles asynchronously in the tile pool. Decoding of compressed texture data is implemented on GPU to minimize texture transfer. The Texture Producer can also generate procedural textures. Our system is transparent to the user, and the only parameter that must be supplied at runtime is the current viewpoint. No modifications of the mesh are required.

We demonstrate our system on large scenes displayed in real time. We show that it achieves interactive frame rates even in low-memory low-bandwidth situations.

Key-words: real-time rendering, high-resolution textures, GPU, progressive loading

Gestion de texture unifiée pour géométrie arbitraire

Résumé : Les jeux vidéos et les simulateurs utilisent couramment des textures très détaillées, dont la taille cumulée est souvent bien plus grande que la taille de la mémoire vidéo.

Les textures peuvent être chargées progressivement, mais charger et transférer dynamiquement ces grandes quantités d'informations dans la mémoire vidéo entraîne des délais de chargement et dégrade les performances. La gestion de la mémoire vidéo dédiée aux textures est donc devenue un problème important pour les applications interactives. Ce problème a été en partie traité pour le cas particulier du rendu de terrain. Cependant il n'existe pas de méthode pour la gestion de textures dans le cas général.

Nous proposons un système, implémenté sur les cartes graphiques actuelles, permettant d'unifier les approches classiques destinées à réduire le coût des transferts mémoire: chargement progressif, compression de textures, stratégie de cache. Pour cela nous introduisons un nouvel algorithme, exécuté par la carte graphique, pour résoudre la difficulté majeure de détecter les parties d'une texture nécessaires pour un rendu.

Notre système est basé sur trois composants manipulant une mémoire stockant des morceaux (rectangles) de texture. Cette mémoire est appelée "tile pool". Le premier composant, la "Texture Load Map", détermine pour chaque image les morceaux de texture nécessaires (position et niveau de détail) pour effectuer le rendu depuis le point de vue courant. Le second composant, le "Texture Cache", gère la mémoire de texture. Enfin, le "Texture Producer" charge et décode les morceaux de texture de manière asynchrone dans la mémoire. La décompression des données de texture est effectuée sur la carte graphique pour minimiser les transferts. Le "Texture Producer" peut également générer des textures procédurales.

Notre système est transparent pour l'utilisateur, le seul paramètre nécessaire est le point de vue courant. Aucune modification de la géométrie n'est nécessaire.

Mots-clés : rendu temps réel, textures haute résolution, GPU, chargement progressif

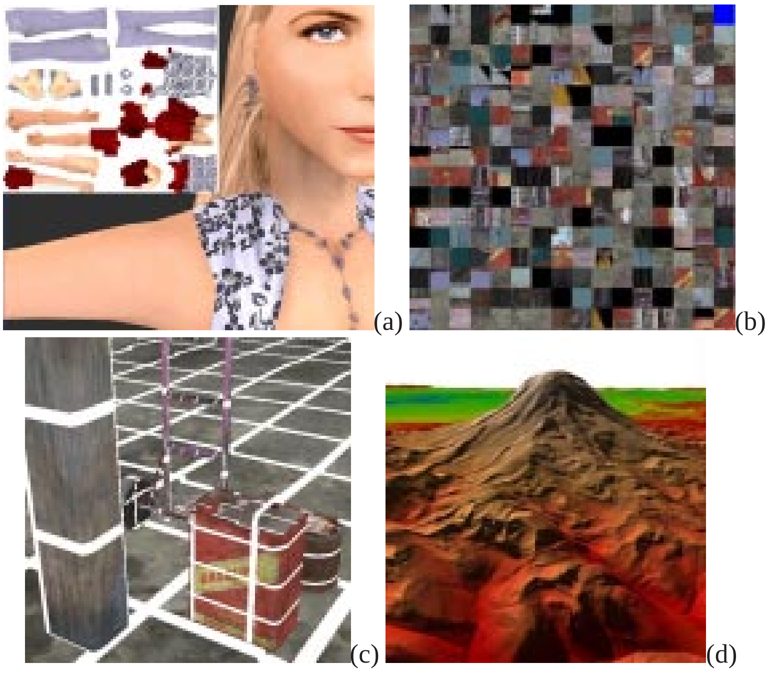


Figure 1: **(a)** in red/dark: parts of the texture atlas detected as needed for the current viewpoint. **(b)** The texture is loaded as tiles in GPU memory for rendering (8MB). **(c)** Our loaded texture tiles mapped onto the scene (whole texture is 128MB). Note that geometry is not modified: our system works in texture space. **(d)** Real-time fly over a terrain mapped with a 1GB texture.

1 Introduction

Textures are a convenient way to add details in images without increasing the geometric complexity. Nowadays video games and interactive applications commonly use multiple layers of highly detailed textures on objects.

Thus, large amount of texture data needs to be loaded and transfered into the GPU memory for rendering. This can lead to noticeable loading delays if texture data is stored on a slow mass media storage (e.g. a hard drive or Internet server). Moreover, the size of this texture data often exceeds the size of the physical memory available on hardware renderers. In this case texture loading must be done dynamically in order to load the data needed for rendering the current viewpoint.

Current graphics APIs provide a basic swap mechanism between CPU memory and GPU memory. Textures are treated as atomic resources: even if only a small part of the texture is used, it is entirely loaded in GPU memory. Moreover only textures fitting in GPU memory are handled. This swapping mechanism cannot be used in practice because it results in poor rendering performances.

Each rendered frame does not need all the texture data at full resolution. Indeed only the texture data viewed from the current viewpoint needs to be accessed by the renderer. Since the screen resolution is bounded, less detailed versions of textures can be used for distant geometry. More generally displayed texture resolution does not need to exceed screen resolution. MIP-mapping [Wil83] is a classical solution to adapt the texture levels of detail during rendering. Therefore, texture data are usually stored as a MIP-mapping pyramid.

To benefit from these properties *progressive loading* is commonly implemented: The application starts with partial texture data at low quality and then details are (down)loaded according to the current viewpoint. This is made under the assumption that the rendering quality can be sacrificed in favor of a higher frame rate. The common approach [GY98, CE98, Hut98] is to subdivide the MIP-mapping pyramid levels into regular grids, defining texture *tiles* that will be loaded or cleared on demand.
Usually a priority rule also determines in which order the tiles must be loaded. The size of the tiles is chosen such that each tile can fit into GPU memory. This organization of texture data is depicted in figure 2.

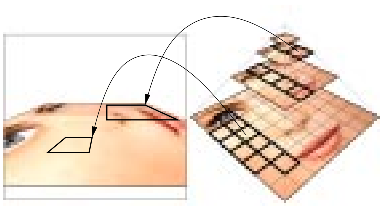


Figure 2: The pyramidal structure of texture space.

However many difficulties arise. Hardware renderers do not include a mechanism to load only part of a texture in memory. Usually to overcome this limitation the geometry is split according to the tiling of the texture. Then the texture is considered as a set of smaller textures mapped on different pieces of geometry. This may produce lot of supplementary geometry and may be time consuming with complex animated meshes, or with meshes handled by geometric level of details algorithm. This approach is usually only practicable with specific geometry such as in the case of terrain rendering. The main difficulty consists of detecting which parts of the texture pyramid are needed for rendering from a given viewpoint. Indeed one needs to compute the visibility of each texture tile – ideally by taking frustum culling and occlusion culling into account – together with the level of detail at which it is used. Since this has to be computed at each rendered frame, an algorithm answering to that problem must also be fast enough for real time applications. It is thus very important to avoid per triangle geometric computations on the CPU.

Many studies have dealt with progressive loading of texture data for the specific case of landscapes rendering. However, these methods do not easily address the problem for arbitrary meshes. Difficulties arise because of the way textures are mapped on the geometry: The texture can be stretched, deformed and the parameterization can be discontinuous. Scenes used in games are often populated with buildings, characters, and so forth. Nowadays they are using very detailed textures and texture management has become mandatory for these scenes.

Other kind of textures, such as procedural textures [Per85], are also of great potential interest for interactive applications. Since procedural textures are defined by functions, they do not suffer from data transfers, but from their computational costs. Procedural textures become affordable on GPUs – thanks to programmability – but they are still difficult to use because of filtering and speed issues. Note that computational cost for procedural textures is equivalent to the loading cost for standard textures: It corresponds to the delay before texture data becomes available for rendering.

Progressive loading is crucial to limit loading delays. Nevertheless other methods are classically used to limit texture data size and memory transfer, such as compression and caching strategies. A texture management system must include these approaches in order to optimize bandwidth usage.
The contributions of this paper are the following:

- A texture management system, transparent for the user, providing progressive loading of high resolution texture in real-time. It supports arbitrary mappings with no modification of the geometry. It includes a memory cache, residing in GPU memory, to benefit from temporal coherency.
- A new hierarchical algorithm to determine visible parts of a texture in real-time.
- A unified model, providing the ability to integrate a procedural texture generator or an hardware accelerated decompresser. In particular we implemented integer JPEG 2000 and Haar decompression schemes in our system.

We demonstrate our system in a wide variety of situations: character rendering, large video game scene, landscape rendering with potential memory and bandwidth limitations.

2 Previous work

We discuss here the previous work on progressive loading of texture data. The main issues are to determine the needed parts of the texture and to efficiently transfer/store data into the renderer memory.

Determining needed texture parts. The *Clipmaps* architecture, presented by Tanner *et al.* [TMJ98], relies on a center of interest – a point and a radius in texture space - provided by the user. It is used to determine which area of the texture must be loaded. The amount of details to load decreases with the distance from the center of interest. Unfortunately this center of interest is difficult to efficiently set for arbitrary meshes since parameterization may be discontinuous. If two distant areas in the texture are seen at the same time, the area of interest would include a very large superset of the actual needed parts. This is commonly the case when a mesh is textured with an atlas [MYV93], where the mapped image is split into several parts. MP-grids [Hut98] do not require the user to specify a center of interest in texture space. Instead the geometry associated with each texture tile is computed. Required texture parts are then determined by a geometrical test. Cline *et al.* [CE98] determines the needed parts of the texture by a geometric computation on each polygon. Polygons must also be split according to the tiling of texture space for rendering. Both methods involve many geometric computations. This makes the use of geometrical level of details or animated meshes difficult. Therefore these approaches – primarily designed for terrain rendering – cannot be easily extended to arbitrary meshes displayed in real time. Dollner *et al.* [DBH00] maintains a hierarchical texture tree associated to the hierarchical model of a terrain geometry. The needed part of the texture are deduced from the level of details selected for the geometry. The technique strongly exploits the correspondence between texture and geometry in the specific case of terrain rendering. Goss *et al.* [GY98] proposed a hardware modification to circumvent the problem of finding the appropriate tiles to be loaded. The number of pixels using a same tile is computed by the hardware. Tiles with the highest counter values are loaded first. Texture is initialized with low resolution data. This approach handles arbitrary meshes and exactly determine texture tiles needed for rendering the current frame. However it requires a deep modification of the texturing hardware and does not easily scale to very large textures.

Transferring and storing texture data. Cache systems are classically used to turn temporal coherence to profit (i.e most tiles are still visible in the next frame and new tiles progressively appear). Cline *et al.*[CE98] caches texture data in video memory by splitting the MIP-mapping pyramid of a large texture in smaller textures (corresponding to the tiles). A caching strategy is described to swap the small textures between main memory and GPU memory. However this organization makes geometry splitting mandatory. Correct filtering cannot either be achieved: Different MIP-mapping levels are stored in separate textures and the discontinuities introduced by the splitting makes linear interpolation difficult. Tanner *et al.* [TMJ98] rely on a specific organization of the texture memory: It stores texture data only around the center of interest. The size of the storage area permits to pre-cache texture data for rendering of next frames. The method proposed in [GY98] focus on the loading latency issue. The whole texture pyramid is supposed to be fit into video memory. However, it is not possible to store the whole texture when dealing with very large textures, a sparse storage is mandatory.

Compression. Compression of images is a classical way to reduce data storage and transfer. Beers *et al.* [BAC96] presented a method based on vector quantization to directly render compressed textures in hardware rendering systems. Nowadays, standard GPUs includes lossy compression schemes such as ST3C to reduce storage cost. However these schemes might suffer from visible artifacts and the compression ratios may not be sufficient with very large texture. Usually the first purpose of GPU implemented compression algorithms is to reduce storage. Decompression is done on the fly during rasterization. However compressing texture data also reduce bandwidth requirement since the texture data is sent in a compressed form. Wavelet compression schemes [Swe96, JPE00] are particularly interesting for this purpose since they offer a hierarchical representation of the texture details.

3 Our Texturing System

3.1 Overview

Our system proposes a unified architecture for texturing meshes with arbitrary mapping. We address the following issues: the efficient determination of the needed parts of the texture, the minimization of texture transfer and storage and the possibility of using complex textures like compressed textures or procedural textures.

Figure 3 depicts the architecture of our system. Our system is updated with the new viewpoint at the beginning of each frame. Once the update is done, the system can be used as a simple texture, through a bind/unbind API. Our system is based on three components interacting with each others. The *Texture Load Map* component indicates which texture tiles are needed for rendering the scene from the current viewpoint. It sends this information to the *Texture Cache* component. Each time

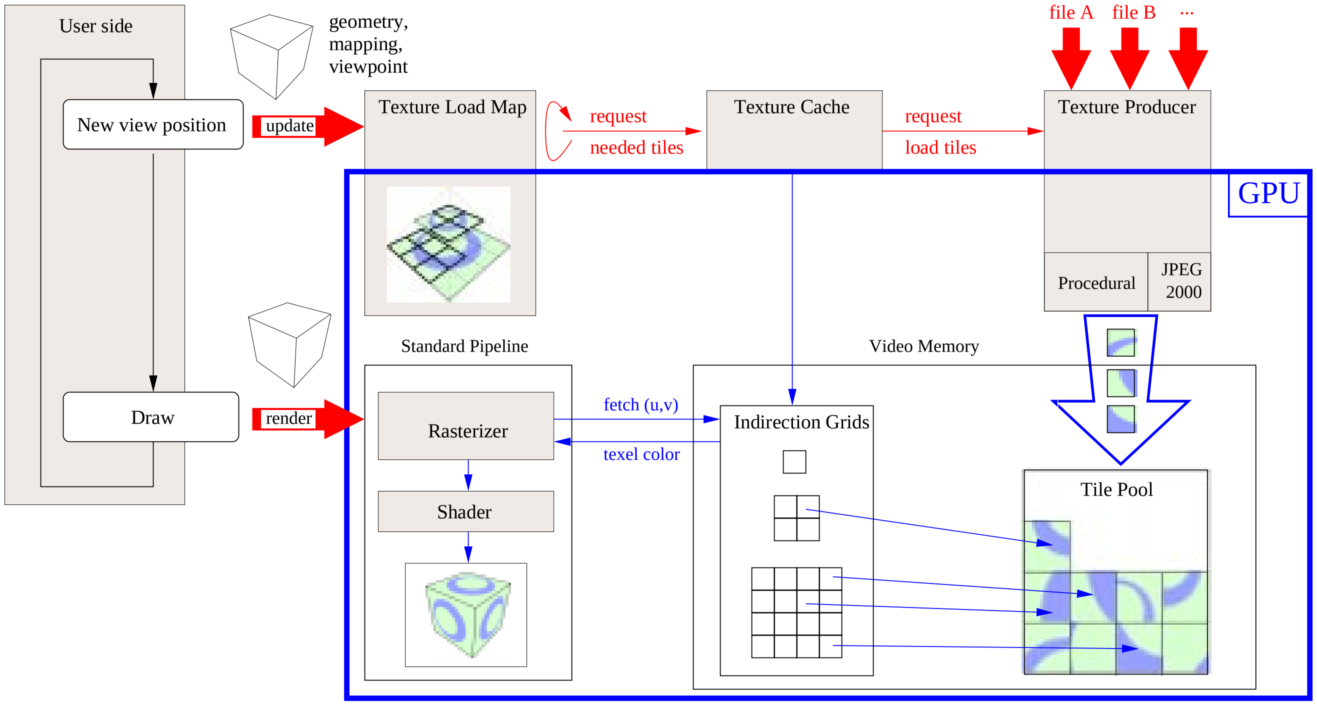


Figure 3: Block diagram of our architecture. The Texture Load Map component sends information about needed texture tiles. The Texture Cache handles storage of texture data. The Texture Producer asynchronously produces texture data for the tiles becoming visible. It reads data from mass storages and decodes it directly into GPU memory. GPU part is outlined in blue.

a tile is needed the Texture Cache checks whether this tile is available in the cache memory (i.e. the Tile Pool). If not it allocates a new slot for the tile and sends a requestto the *Texture Producer* component. The latter generates tile data directly into the Tile Pool. It can be implemented as a texture loader, texture decompresser or a procedural texture generator. Loading is asynchronously performed by the Texture Producer, according to a user–defined bandwidth constraint.

Our architecture is well integrated in the standard pipeline: The application simply sends view-point and geometry updates to the Texture Load Map component. The memory containing texture data – i.e Tile Pool – can be used at *any* time to render a frame since it is stored in GPU memory. The Texture Cache guarantees that stored data are coherent. Missing tiles are not drawn and low resolution version of the texture is used to fill in the gaps.

The main advantage and true novelty of our system is to avoid geometric computations other than rendering – which is performed efficiently by the GPU. Thanks to our new algorithm to implement the Texture Load Map and to the use of indirection textures to store the texture cache, the CPU never has to work directly on the geometry, nor to modify it for rendering.

Our system also speeds up the generation of textures requiring computations, like compressed textures or procedural textures. Usually these types of textures are decoded on the fly during rendering, resulting in poor performance since computation takes place for every rendered pixel, at each frame. Thanks to the Texture Producer component our system enables such textures to be directly cached in GPU memory, thus taking advantage of temporal coherency. Once the data stored in the cache, rendering requires a simple lookup in the Tile Pool. This also decreases bandwidth requirement, since in the case of compressed textures, only compressed data is sent into the GPU memory. With procedural textures no data, apart from the texture procedure, is sent through the bus since the texture is entirely computed by the GPU. Note that the cache must only contain enough information to render one frame, which means that, ideally, it can be no larger than the frame buffer (since we are storing squared tiles and not single pixels, the minimum size of the cache is larger but of the same order of magnitude). Therefore storing uncompressed data in the cache does not result in very high memory requirements, while enabling rendering speed-up.

While our texturing system can handle the whole texture pyramid, GPUs are most efficient with textures of reasonable size not requiring progressive loading. Therefore, we usually use our system only to handle the latest levels of a large texture pyramid. The first levels are stored as a standard texture in the GPU. Our system becomes active only if the higher levels of the texture are used.

The three components of our system are detailed in the following sections.

4 Texture Load Map

4.1 Overview

The Texture Load Map (TLM) component computes which part of the texture is needed for the current view-point. The TLM component computes a map of the texture space containing visibility information for each tile. We store this map as a pyramid of 2D maps. One 2D map is computed for each level of the texture space pyramid. We refer to these 2D maps as *TLM levels*. Figure 5 shows the TLM levels for a simple case. For each texture tile, a TLM level contains a minimum and maximum levels of detail (LOD) value. These LOD values correspond to the minimum and maximum level of our system at which pixels within the tile are using the texture. A LOD value always lies in the interval $[0,k]$ where k is the number of levels handled by our system (see Figure 4). If a tile's level falls within the maximum and minimum LOD values, then the tile is needed for rendering. The TLM also stores a priority value for each tile.

The *children tiles* of a tile T are the four tiles corresponding to the same texture area in the next (more detailed) level of the pyramid. T is then referred to as the *parent tile*. We shall refer to the first texture level handled by our system as the *first handled level*.

We propose a GPU-based algorithm to compute a TLM which performs on arbitrary meshes – possibly animated – with any parameterization. It is based on a hierarchical approach and produces a conservative estimation of visible texture parts. This algorithm is easy to implement and performs in real-time. First we describe how to compute a given TLM level. Then we describe our algorithm for computing a whole TLM.

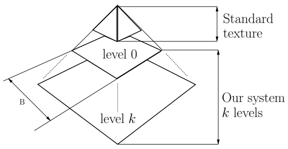


Figure 4: Our system handles the latest levels of the texture pyramid.

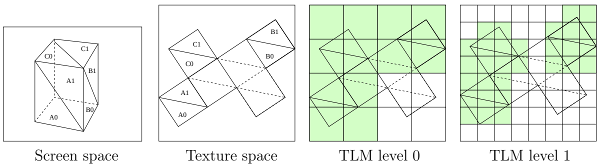


Figure 5: The Texture Load Map component computes the set of visible texture tiles, marked in green/gray in the TLM levels.

4.2 Computing a TLM level

The key idea of our algorithm is to render the geometry into texture space. This is done by rendering the geometry using 2D texture coordinates of the vertices instead of their world coordinates. The rendering resolution is chosen such that one rendered pixel corresponds to one tile of a TLM level. One rendering computes one TLM level. When a pixel is rasterized in the TLM level at coordinates (i,j), it implies that the triangle being rasterized is textured using some part of the texture covered by the tile at (i,j). This is illustrated by Figure 6. Note that it does not imply that the tile is actually *needed*, this will depend on the selected level of detail. During rasterization of the TLM level, we also have access to screen coordinates and world coordinates interpolated from triangle vertices (screen space coordinates are computed in a vertex program).

Frustum culling and back face culling. Frustum culling is performed by culling triangles in texture space according to the current viewpoint. This is achieved with the clipping registers available on recent nVIDIA GPUs. Only the geometry within the camera view frustum is rendered in the TLM level. This is depicted on figure 7.

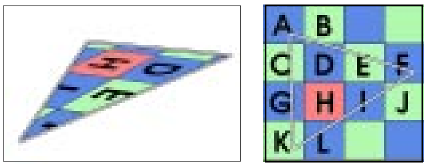
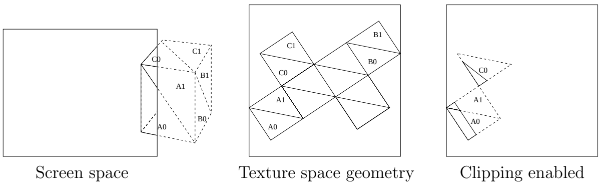


Figure 6: *Left:* Textured triangle rendered in screen space. *Right:* Same triangle rendered in TLM level of resolution 4 × 4. Texture coordinate of vertices are used for rendering. Rasterized pixels each correspnd to one tile. Rasterized tiles are marked by a letter.

Backface culling requires a special treatment: GPUs automatically handle back–facing triangles, but only by taking into account vertex position in rendering space. In our case – since we are rendering in texture space – the vertices positions are not their screen space position but their texture space coordinates: Automatic back face culling according to the screen space cannot be performed. Our approach is to use the far clipping plane to achieve back face culling. Recall we are rendering in a 2D texture space: The near and far clipping plane were not taking into account until now. We compute the z coordinate of the vertices during rendering in the TLM level as $hpos.z = dot(nrm, vv) - 1.0$ where $hpos.z$ is the z coordinate of the resulting vertex (as computed in the TLM *vertex program*), nrm the normal associated to the vertex, and vv the normalized view vector. The far plane is positionned at $z = -1$. Vertices are thus culled if $dot(nrm, vv) < 0.0$ which corresponds to the operation required by backface culling. Note that if per-vertex normals are used, the triangle will be split at the iso-value $dot(nrm, vv) = 0.0$.

The same geometry is used for rendering both the TLM and the camera view. Since the same clipping is performed for both renderings, the geometrical cost of a TLM level is equivalent to the geometrical cost of rendering from the current viewpoint. Moreover, only the pixels corresponding to triangles within the camera view frustum are produced, which reduces per-fragment computations in the TLM level.



Screen space

Texture space geometry

Clipping enabled

Figure 7: Texture space triangles are clipped with respect to the current viewpoint.

Conservative result. In order to produce a conservative result, our algorithm must use a rasterizer such that any pixel containing a piece of geometry is rasterized. We will refer to such a rasterizer as a *conservative rasterizer*. Indeed without this property small isolated triangles – in texture space – could be missed. As a consequence some parts of the texture would not be loaded when required.

We emulate a conservative rasterizer by enabling antialiased polygons or by rendering the model in antialiased wire-frame mode on top of the polygons. The rasterizer then produces all fragments containing geometry [Ope03].

Information from triangles using a same texture tile have to be combined to compute the maximum and minimum LOD values. This occurs when different triangles use a same texture tile. To combine these information we render the TLM level with a *max* blending mode. The fragment program computing the LOD value $v \in [0,1]$ of a triangle outputs both v and $1.0 - v$, thus allowing for computation of the maximum and minimum LOD values over multiple triangles.

Texture coordinates range. Texture coordinates outside the $[0,1]^2$ texture range (floors, wallpapers) can be handled by keeping only fractional coordinates of pixels rasterized in texture space. Note that today's GPUs do not support such operations. Therefore our current implementation does not handle texture wrapping.

Computing the required levels of detail. During rasterization of triangles in the TLM level, we need to compute which level of our system is used to texture the triangles in screen space. This information will be encoded in each pixel – i.e. tile – of the TLM level. It corresponds to the LOD value at which the triangle being rasterized is using the tile. Recall that LOD values of all the triangles using a same texture tile are combined to compute the minimum and maximum LOD values (see above, *Conservative result*).

The level used to texture can be deduced by computing how many screen pixels lies between two neighboring tiles of the TLM space: This is given by the derivatives of the screen space coordinates with respect to the texture coordinate system. The finite differences operators (`ddx` and `ddy` instructions of nVIDIA hardware) can compute these derivatives during rendering of the TLM level. However this suffers from precision issues, which results in wrong texture tile selection.

We propose another approach based on the standard MIP-mapping algorithm implemented in GPUs. We create a 2D MIP-mapped texture – referred to as *the LOD texture* – with a resolution of $2^{k-1} \times 2^{k-1}$ pixels where k is the number of texture levels handled by our system. The LOD texture has k MIP-mapping levels. Each MIP-mapping level of the LOD texture has a color corresponding to the index of the level of our system to be selected: Every pixels of level i have a color $R = G = B = i$ (the LOD texture is actually a luminance texture, storing only one value per pixel). This is depicted Figure 8. During the rendering in texture space we retrieve the color of the pixels from the LOD texture. The LOD texture is accessed with the screen space coordinates interpolated from triangle vertices. Since the LOD texture is filtered by the GPU, the resulting color corresponds to the MIP-mapping level selected by the texture unit. This color *is* the index of the level of our system that is used for rendering (the LOD value).

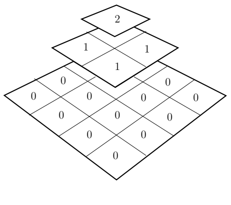


Figure 8: The LOD texture for $k = 3$ levels handled by our system.

The GPU selects the MIP-mapping level according to the LOD texture resolution and the resolution of the TLM level in which rendering occurs. However we need to compute the required MIP-mapping level for the texture handled by our system. We thus shift the MIP-mapping level selection of the LOD texture. Let M^2

be the resolution of the TLM level, B^2 the resolution of the first texture level handled by our system (see Figure 4); L^2 the resolution of the LOD texture. Let $l \in [0,1]$ be the coordinate used to access the LOD texture, $x \in [0,1]$ be the texture space coordinates and $s \in [0,1]$ the screen space coordinates. During rendering in the TLM level, the LOD texture is seen at full resolution if $\frac{dl}{dx} = \frac{M}{L}$. We would like this to correspond to the selection of the first level of the texture handled by our system. The first handled level must be selected for $\frac{ds}{dx} = \frac{B}{S}$. In other words $\frac{dl}{ds} = \frac{S}{B} \times \frac{M}{L}$. Therefore the screen space coordinate s must be scaled by $\frac{S}{B} \times \frac{M}{L}$ before accessing the LOD texture (recall each level of the LOD texture has a uniform color - the color is only given by the selected MIP-mapping level). Figure 9 illustrates level selection with the LOD texture.

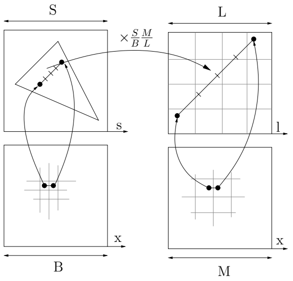


Figure 9: *Left, top:* The line on the triangle is four pixel long on the screen. *Bottom:* The line is textured using two pixels of the first level handled by our system. In this case $\frac{ds}{dx} = 4\frac{B}{S}$ and the third level (index 2) handled by our system must be selected to render at full resolution. *Right, bottom:* Rasterization of the same line in the TLM level of resolution M. *Top:* The line is textured with the LOD texture. The screen coordinates are multiplied by $\frac{S}{B} \times \frac{M}{L}$ before accessing the LOD texture. The GPU uses the last MIP-mapping level of the LOD texture, containing the index 2 ($\frac{dl}{dx} = 4\frac{M}{L}$). This selects the third level handled by our system, which is the expected result.

The drawback of this approach is that the LOD texture has a resolution of $2^{k-1} \times 2^{k-1}$, which may waste some texture memory if many texture levels are handled by our system. This limitation could be overcome if the GPU could give access to MIP-mapping level computation done in texture units.

Occlusion culling. Our algorithm, as presented above, offers a conservative estimation of the optimal set of tiles needed for rendering. This estimation is already quite good since a tile is marked as needed if and only if at least one front facing triangle using the tile is within the camera view frustum.

However occlusion culling is not taken into account. In complex scenes a large part of textured area can be hidden by occlusion. This is, for example, the case in urban environment scenes. It is therefore very important for our texturing system to be able to handle occlusion culling. To avoid complex geometrical computation we rely on the *shadow buffer* algorithm [Wil78]. We check with a lookup in a low resolution depth map rendered from the current viewpoint whether the point in world space coordinate associated with a pixel rasterized in the TLM level is occluded. Unfortunately this cannot ensure a conservative result due to aliasing of the depth test and to the fact that the tested point actually corresponds to a whole area of the triangle being rasterized. If this point is considered occluded, the part of the triangle textured by the tile is entirely considered as occluded, which may be wrong. For applications where rendering quality can be sacrificed for higher frame rates, this approach is suitable. If a conservative result is mandatory, this approach can nevertheless be used to load non occluded tiles with a higher priority.

4.3 Computing a whole TLM

The algorithm for computing a whole TLM benefits from the following properties:

- A map of the TLM can be deduced from any map of higher resolution: The minimum (resp. maximum) LOD value of a tile can be computed as the minimum (resp. maximum) over the LOD values of its children tiles. The result remains conservative by construction.
- Only a small area of the TLM map of higher levels of detail is used from a given viewpoint: The screen resolution limits the amount of texture visible at once. However the *active area* is not always continuous. For instance if a texture atlas is used, non-neighboring parts of the texture can be simultaneously seen from a given viewpoint (e.g the texture of the eyes of a character can be stored far from the texture of its face). Thanks to conservative properties, the active area of TLM levels of higher levels is always included in the active areas of TLM levels of lower levels.

From these two properties we propose the following algorithm:

```
Set active area to the whole texture space
Render TLM level (lowest level)
Compute the active area
Limit rendering to the active area
for (i=lowest level+1; i<=hightest level; i+=n)
  Render TLM map (i+n)
  Compute the active area
  Limit rendering to the active area
  for (j=i+n-1; j>=i; j--)
      Compute TLM map (j) from TLM map (j+1)
```

This algorithm involves a rendering every *n* levels. The rendering is restricted on the active area estimated from previous levels. Therefore, it involves less geometric and fragment processing than computing explicitly all TLM levels. The value of *n* controls the tradeoff between precision and performance (for our applications we set *n* to the number of levels). Each rendered TLM map has to be retrieved from video memory for the texture loading process. This transfer is slow (GPUs architecture are not designed for these read backs). However, thanks to our algorithm we only retrieve the active parts of the TLM levels.

4.4 Generating tile requests

Tile requests are generated from the TLM. Since the TLM contains information for all the tiles of the texture pyramid, it can be computationally extensive to parse it entirely. Once again, we take profit from the hierarchical structure of the TLM.

Each TLM level contains the maximum and minimum LOD value for a tile. A tile is needed if the tile level is between the minimum and maximum LOD values stored in the TLM for the tile. If a tile at a given level is not needed, it implies that none of its children tiles are needed. Indeed a tile is not needed if the maximum LOD value for the tile is below the actual tile level in the pyramid. None of the children tiles can have greater LOD value, since they share the same pixels of the texture. Note than since tiles are actually MIP-mapped in the Tile Pool (see section 5), a tile does not need to be flagged as needed if its children tiles are themselves needed. This can be checked easily by examining the LOD values of the children tiles.

We first entirely parse the least detailed level and compute the list of needed tiles for this level. To parse the second level, we only examine children tiles of needed tiles of the first level. The resulting list of needed tiles is then used to parse the third level. We recursively apply this process. The output is the set of tiles needed for the current frame.

Requests are generated by comparing the set of tiles for the current frame with the set of tiles for the previous frame. Requests are generated for tiles that were not needed at the previous frame and are now needed. Tiles that are no longer needed are flagged in the Texture Cache – these tiles will be the first to be erased if memory lacks.

5 Texture Cache

The purpose of the Texture Cache component is to handle storage of needed tiles for the current rendering and to implement a least recently used (LRU) cache of tile data. Since we need fast rendering, the cache memory must be in video memory.

We extend the method of [KE02] to implement a sparse storage of the MIP-mapping pyramid. Cached tiles are packed into a texture referred to as *Tile Pool*. Each level of the pyramid is covered by an indirection grid. Each cell of the indirection grid covers exactly one tile, and contains a pointer to a tile in the Tile Pool. This is illustrated on figure 3. Once the Texture Cache receives a tile request for a non-cached tile, it performs the following operations: reserve a slot for the tile in the Tile Pool, set the tile pointer of the indirection grid to null (i.e missing tile), send a request to the Texture Producer. As soon as the texture data is generated, the Texture Producer send an acknowledge and the pointer in the indirection grid is updated. In order to minimize transfer, indirection grid updates are sent once in GPU memory at the beginning of each frame. Only a rectangular area containing the updates is transfered.

Memory thrashing. In some cases the cache may not be large enough to store all the data selected for the current frame. When this occurs, the Texture Cache first discards the tiles with the lowest priority in the current frame (see section 6). Missing tiles are rendered with a low resolution version of the texture.

Rendering the cached data. The texture data stored in the Tile Pool is rendered by the GPU. To achieve high quality rendering, MIP-Mapping must be performed. We store not only a tile but its whole MIP-mapping pyramid. This increases memory usage of the cache memory by 33%. Note that parent tiles are not loaded if their four children tiles are also loaded to avoid storing multiple times the same information. Computation of this MIP-mapping pyramid is performed either by the GPU or by the Texture Producer component for procedural textures. As explained in [LN03], correct filtering of indirection textures cannot be automatically handled at tiles boundaries and must be explicitly programmed. Note that this task could be greatly simplified by a native hardware support of tile based textures.

6 Texture producer

The purpose of the Texture Produce is to generate the needed texture tiles that are not already in the Tile Pool. It receives render requests from the Texture Cache. These requests are added to a priority queue and processed asynchronously according to a user define bandwidth limitation.

The CPU part of the Texture Producer – implemented as a loading thread – is in charge of retrieving data from slow mass media storage. Data needed to generate texture tiles are uploaded from CPU memory to GPU memory as textures. The Texture Producer then renders tile data directly into the Tile Pool. This is performed in hardware using a *render to texture* operation with a viewport covering the destination tile. A quad is rendered, using a special fragment program in charge of *decoding* texture data. The input of this fragment program are the coordinates of the tile within the texture pyramid, its resolution and the data loaded from mass media storage for the tile (stored as textures).

The standard Texture Producer simply copies the uploaded texture tiles into the Tile Pool. A more complex Texture Producer – like a texture decompresser – would decode the data during rendering into the Tile Pool. The transfer of decoded texture data is thus only done within GPU memory, which minimizes transfer between CPU and GPU memory.

To illustrate the flexibility of our system we implemented two wavelet based decompresser: simple Haar wavelets and lossless JPEG 2000 integer wavelets. Wavelet decompressers are especially well suited for progressive loading: Only details need to be loaded to increase resolution of the data already in cache, thus reducing loading and bandwidth requirements. Details can also be sent in most relevant first order very easily. Implementing these decompressers on a GPU, while technically challenging, is rather straight forward. We rather focus on the core of our texturing architecture and will not give implementation details about these texture producers.

Prioritized loading . Visualization of large textures during progressive loading can suffer from a lack of data. Using a priority rule alleviates this problem by loading the most relevant parts of the texture first. The TLM component computes a priority based on visual criteria during the rendering of a TLM level: distance from screen center, and distance from the viewpoint. The Texture Producer component proposes a data criteria based on a user defined priority value in texture space, or on wavelet detail coefficients. Final priority for a tile is computed as a weighted sum of both criteria. Tile requests are sorted with respect to their priority value before being processed by the Texture Producer.

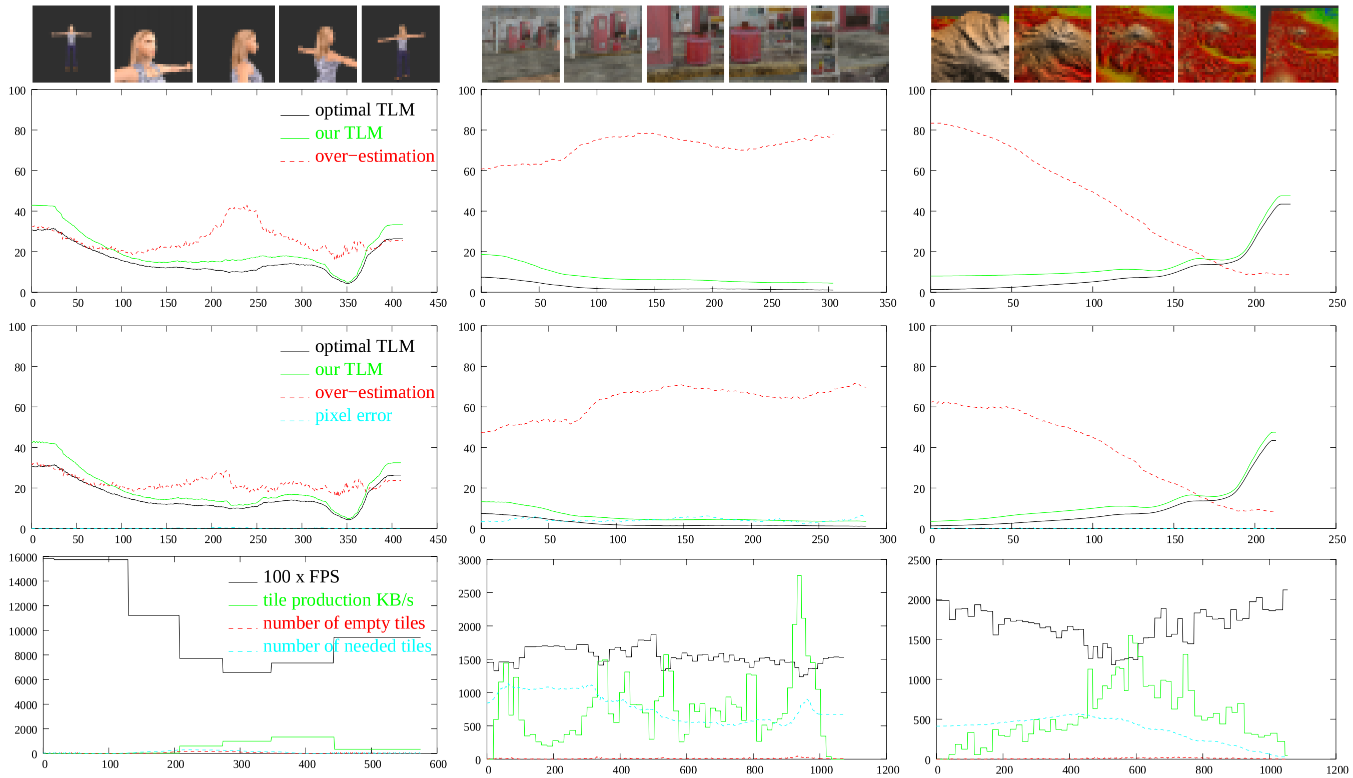


Figure 10: *From left to right:* Character model, Gas Station scene and Puget Sound terrain. *From top to bottom:* measure of TLM quality, measure of TLM quality with occlusion, TLM efficiency and overall system performance. The fly–overs are shown in the accompanying video.

7 Results

We have tested our system in various situations: landscape rendering, character meshes and architectural scenes. We describe below how our system can be used in these cases.

In all the examples presented, we let the GPU cope with the low resolution levels of the MIP-map pyramid (typically, below 1024^2 pixels). Our system handles the more detailed levels of the texture. Benchmarks are done on three models (landscape, character, architectural scene) with a fixed camera path. Measurements are done on a Pentium 3GHz with a nVIDIA GeForce FX 5950.

Landscape. Since our system handles arbitrary meshes, it performs well on large textured landscapes. We use data of the *Puget Sound* terrain (courtesy of USGS and The University of Washington). The texture resolution is 16384×16384 RGB pixels (1GB with MIP-mapping pyramid). The four last levels of the texture pyramid are handled by our system. Tile resolution is 64×64.

Characters. Characters are our first example of arbitrary meshes. Characters are classically textured using an atlas, often created by artists. Our TLM algorithm performs well in this case. It is interesting to note that our system never loads the unused part of the texture atlas. Nevertheless optimizing the atlas is important to reduce storage on mass media and optimize resolution usage in the texture. Our system is of course compatible with any re-parameterization scheme done in preprocessing since it performs on arbitrary mapping. The texture used is 2048×2048 RGB. The two last levels are handled by our system. Tile resolution is 32×32.

Architectural scene. This type of scene is usually composed of many textures applied on many objects. To handle this case in our system, we create a unique texture for the scene: We pack all textures into one big texture and we update the texture coordinates of each object. Note that this packing do not need to be efficient: thanks to our TLM algorithm empty spaces will never be loaded. The update of the texture coordinate of each object can be made transparent to the user through a binding API making use of the texture matrix. We demonstrate this on the nVIDIA Gas Station demo (models and textures courtesy of NVIDIA Corporation). The scene consists of about 26,000 triangles and the amount of textured data is 128 MB. The three last levels of the texture are handled by our system. Tile resolution is 32×32.

7.1 Texture Load Map

Performance. The TLM performs fast enough for real time applications. We measure both frame rate and tile requests generation, with only TLM computation active. This includes read back of TLM data. For the character, average TLM frame rate is 135 FPS and CPU cost of tile requests ranges from 0.02 ms to 0.6 ms. For the Gas Station scene, average TLM frame rate is 36 FPS and CPU cost ranges from 2.7 ms to 6.0 ms. For the Puget Sound terrain, average TLM frame rate is 90 FPS and CPU cost is below 1 ms.

Quality of estimation. We estimate the quality of our TLM algorithm by comparing the selected set of tiles with the optimal set of tiles. The optimal set of tile is computed by rendering the current frame with a fragment program displaying texture coordinates in the frame buffer. We then read back the frame buffer, parse the texture coordinate, and compute the optimal set of tiles. This is a very slow process due to the costly read back (whole frame buffer) and the parsing of screen pixels.

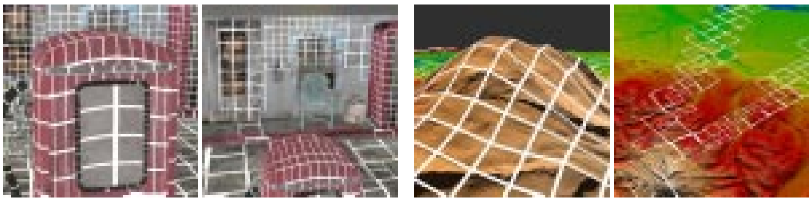


Figure 11: Close view of the Gas Station scene and the Puget Sound terrain. First image is the current view point, second image shows a different viewpoint of the same data. In both cases, occlusion culling avoids loading texture data hidden by close geometry.

Figure 10 summarizes the results, with and without occlusion culling enabled (first and second rows). The two first curves show the percentage of the whole texture data selected by the optimal TLM and by our TLM algorithm. The over-estimation is given in percentage of tiles wrongly selected as needed by the TLM algorithm, *within the set of selected tiles*. The error is the percentage of pixels not textured correctly in the final image. This only occurs with occlusion culling since the result is no longer conservative. The estimation of our TLM algorithm closely follows the optimum. Over–estimation increases in occluded scenes (Gas Station scene and Puget Sound Terrain). Notice how over–estimation decreases at the end of the Puget Sound terrain fly-over: thanks to the altitude there is almost no occlusion culling, and the TLM is almost equal to the optimum. The TLM algorithm with occlusion culling reduces over–estimation. In the character fly-over, the peak in over-estimation (around frame 230) is removed by occlusion culling: it corresponds to the time at which the arm of the character is hidden by its head.

7.2 Rendering under memory constraint

Under memory constraint, our system concentrates resolution around the user “look–at”. This is done thanks to the tile loading priority (see section 6). Figure 12 shows a view of the nVIDIA Gas Station obtained under strong memory constraint (constraint is 11MB, texture data is 128MB).

7.3 Performance of the whole system

Figure 10 (third row) shows the performance of our system for the three test scenes. In the three cases, our system manages to keep a real-time frame rate. Note how tile rendering influences performance. The number of empty tiles is kept very low, which ensures good rendering quality since the TLM is conservative.

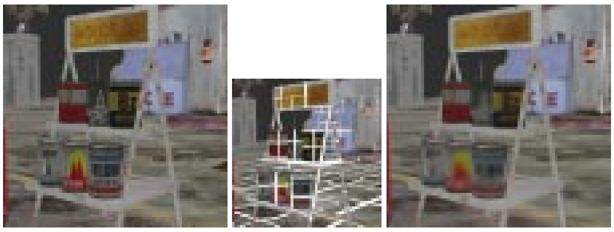


Figure 12: Two viewpoints rendered with 11 MB. Left picture is obtained with our system, right picture is obtained with a resized version of the texture. Details are preserved by our system.

7.4 Latency, bottleneck and limitations

The TLM algorithm performs on the GPU, which implies that its result must be uploaded in CPU memory for processing. In practice the latency introduced by this operation is difficult to measure. However it does not result in poor performances, as demonstrated by the TLM performance.

The current bottleneck of our system mainly concerns rendering. First, we need to use a special shader to render with correct filtering. This shader involves only simple operations but is quite long, thus increasing rendering cost. This could be solved by an hardware support of tile based textures. Second, we need to render each level separately, one on top of the other (only active levels are rendered). This implies to render the geometry multiple times. This limitation will be overcome in near future by using dynamic branching in a fragment program accessing the texture levels.

The maximal size of the texture that can be handled by our implementation is currently limited. Our Texture Cache implementation requires one indirection grid by TLM level. The size of the indirection grid of one level is the size of the texture at this level divided by the size of the texture tiles. For very large textures the indirection grid can still be too large to be stored in video memory. In practice this occurs when resolution of an indirection grid is greater than 4096^2. Assuming tiles of 64^2 size, this gives a maximal texture resolution of 262144^2. For larger resolutions, a solution to this problem would be to implement hierarchical indirection grids.

8 Conclusion

We have presented a new system for texturing arbitrary meshes with high resolution textures in interactive applications. Our algorithm to efficiently compute visible parts of a texture allows to save bandwidth and storage by considerably reducing the amount of data that needs to be transfered. Despite its GPU-based design, the component-based architecture allows for flexibility. Thanks to the concept of Texture Producer, different texture types are made available in hardware, including compressed textures and procedural textures. It is a significant step towards virtualized memory for GPUs.

References

[BAC96] BEERS A. C., AGRAWALA M., CHADDHA N.: Rendering from compressed textures. In *SIGGRAPH 1996* (1996), pp. 373–378.

[CE98] CLINE D., EGBERT P. K.: Interactive display of very large textures. In *Vis'98* (1998), pp. 343–350.

[DBH00] DOLLNER J., BAUMMAN K., HINRICHS K.: Texturing techniques for terrain visualization. In *Vis'2000* (2000), pp. 227–234.

[GY98] GOSS M. E., YUASA K.: Texture tile visibility determination for dynamic texture loading. In *Graphics Hardware '98* (1998).

[Hut98] HUTTNER T.: High resolution textures. In *VisSym '98* (1998).

[JPE00] JPEG: Iso/iec jtc 1/sc 29/wg jpeg 2000 part 1 final committee draft version 1.0. 1.29.15444, 2000.

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Graphics Hardware '02* (2002).

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

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

[Ope03] OPENGL SGI: *The OpenGL Graphics System: A Specification (version 1.5). Mark Segal and Kurt Akeley*. Sillicon Graphics, Inc., 2003, ch. 3 - Rasterization.

[Per85] PERLIN K.: An image synthesizer. In *SIGGRAPH'85* (July 1985), vol. 19(3), pp. 287–296.

[Swe96] SWELDENS W.: The lifting scheme: A custom-design construction of biorthogonal wavelets. *Appl. Comput. Harmon. Anal. 3*, 2 (1996), 186–200.

[TMJ98] TANNER C. C., MIGDAL C. J., JONES M. T.: The clipmap: A virtual mipmap. In *SIGGRAPH 1998* (1998), pp. 151–158.

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. In *SIGGRAPH 78* (1978), pp. 270–274.

[Wil83] WILLIAMS L.: Pyramidal parametrics. In *SIGGRAPH'83* (1983), pp. 1–11.

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

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

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