A GPU Tile-Load-Map architecture for terrain rendering: theory and applications

Yacine Amara · Xavier Marsault

© Springer-Verlag 2008

Abstract This paper describes a robust, modular, complete GPU architecture—the Tile-Load-Map (TLM)—designed for the real-time visualization of wide textured terrains created with arbitrary meshes. It extends and completes our previous succinct paper Amara et al. (ISVC 2007, Part 1, Lecture Notes in Computer Science, vol. 4841, pp. 586–597, Springer, Berlin, 2007) by giving further technical and implementation details. It provides new solutions to problems that had been left unresolved, in the context of a joint use of OpenGL and CUDA, optimized on the G80 graphics chip. We explain the crucial components of the shaders, and emphasize the progress we have proposed, while resolving some difficulties. We show that this texturing architecture is well suited to current challenges, and takes into account most of the distinctive aspects of terrain rendering. Finally, we demonstrate how the design of the TLM facilitates the integration of geomatic input-data into procedural selection/rendering tasks on the GPU, and immediate applications to amplification.

Keywords Terrain rendering · GPU architecture · Level of detail · Data amplification · Seed model

1 Introduction

In the context of modern 3D applications, where CPU overhead is a common drawback, our TLM architecture can be viewed as the first full GPU system capable of rendering vast terrains in real time. It is well suited to the current challenges of their display, and the massive use of recent GPUs as computation boosters. It performs full-texture selection, streaming and rendering, leaving maximum CPU computing time available for other demanding processes such as artificial intelligence tasks. It requires various data sets, structures and adapted algorithms for the different computation steps on the GPU.

This architecture works best on the G80, though it can also function on older GPUs, with some minor restrictions and implementation variations. Working with arbitrary terrain meshes, we will not particularly deal with algorithms that procedurally model, build or render them, but will use one particular example for demonstration purposes. For more information, see [33], a recent exhaustive overview of multiresolution algorithms for terrain mesh generation over the last decade.

The TLM framework can be easily extended to on-the-fly terrain amplification, including fine-scale ground details and added plant/mineral objects. These features are important in many fields dealing with terrain rendering (e.g. GIS, scientific purposes, training simulations, flight simulations, civil engineering, video games, military systems and virtual reality).

This paper starts with general considerations about large texture processing, simplified enough for a non-specialist first reading. It includes a short overview of previous work (Sect. 2.1) followed by an outline of our technique (Sect. 2.2). We then describe the different aspects of the TLM, and give details on our latest implementation.
(Sect. 3). Finally, we propose a complete application where plant covering is automatically suited to terrain characteristics: we explain it in detail in Sect. 4. The results are presented and discussed in Sect. 5, followed by a conclusion and a review of prospects.

2 Large-area texture management and visualization for terrain

2.1 Previous work

In order to display expansive terrains in real time, a number of studies have focussed on three major types of technique required for texture management and mapping:

- specific structures to organize a number of available texture (and geometry) sets
- complex level-of-detail computation
- implementation of memory-resident or out-of-core database handles.

However, little work has been done on such techniques, compared to the impressive amount of work that has been devoted to geometrical aspects of terrain.

Even before graphics chips became programmable, researchers were facing a major problem: how could massive textures be fitted into a small memory space? Even now that modern terrain navigators handle thousands of GB of geo-referenced textures, GPU memory barely reaches one GB. And this gap is likely to persist into the future, as needs in terms of database resolution always grow beyond hardware capabilities.

This problem began to be solved by previous work which revealed the importance and the need for hierarchical data structures of level of details: quadtree [19], octree [26, 28] and clipmap [41]. They have proposed efficient main-memory implementations with corresponding CPU algorithms, and have been widely used to provide the textures strictly necessary to display the current viewpoint [5, 19, 41] and facilitate data storage and streaming. View-frustum culling for texture was always performed by the CPU during a traversal of these structures, often associated with linked geometry data sets. But their usage always imposed constraints on the ground mesh. Splitting it into many regular or non-regular pieces with different numbers of bound textures per draw call was a popular solution, when programmable texture access and control in shader was not available, or not fully implemented. For example, [19, 23, 37] retain a dependency between texture and geometry for quality control and tuning.

Mutual independence of geometry and texture management was achieved in 2002 with a static GPU-based coordinate translator for virtual textures [24]. In 2004, previous techniques using texture-clipmaps [19, 41] were transferred to the GPU and heavily boosted [3] to deal with mesh geometry and texture. However, a recent approach (2007) shows that clipmaps can be hardware-independently implemented with equivalent efficiency [39]. In 2006, A.E. Lefohn et al. described “Glifit,” a C++ generic template library for defining and using complex and random-access data structures for graphics processors (e.g. list, quadtree and octree) and adaptive grids, which makes it possible for applications to efficiently support high database resolution [28]. Although their authors describe it as powerful (which it is!) and simple, this library seems to be designed to handle quite complex problems. Finally, a recent paper [31] proposes an elegant, simple, effective alternative to geometry clipmaps.

Level-of-detail management—particularly on the GPU—has become an important challenge in the context of managing huge sets of textures. Even if most recent navigators succeed in worldwide streaming and in displaying large parts of the earth, they suffer from unaesthetic visual popping artifacts and mesh inconsistencies, due to a lack of care in geometric database handling. And in order to balance the competing requirements of realism and frame rate for high-speed fly-navigation, speed control over the LOD selection seems to be needed; but it is almost never found in applications.

Memory-resident, or out-of-core database handling, is used in conjunction with efficient caching techniques and compression mechanisms to dynamically download them from virtual memory (main memory, hard disk, distant servers) to the main system memory (CPU or GPU) with minimal latency [23, 31, 37, 39]. In order to reduce the amount of on-chip bandwidth required to transfer texels—generally by a factor of ×2 to ×6—texture (de)compression is now widely available on graphics hardware at rendering time. And current texture formats such as DXT make it difficult to tell the difference visually.

2.2 Outline of the TLM architecture

Our approach was mainly inspired by the recent S. Lefebvre’s work on texture [26], which describes how to deal with arbitrary meshes/scenes and build a basic streaming process for progressive and dynamic texture loading onto a GeForce6 graphics card. This framework, working in “virtual texture space,” was used to traverse tile structures by carrying out complex, expensive computations on the GPU. We found the main ideas of this work promising for large terrain displays, and perhaps also for planetary systems. With this in mind, we improved it in many respects (Sects. 3, 4), so that it would work efficiently on the G80 chip. We have also provided simple and powerful solutions to a number of issues involved in wide terrain visualization and amplification.

Our TLM architecture is dedicated to the real-time texture visualization of large-scale terrains. It is based on three
powerful modules. The first two are written in GLSL and associated with a GPU rendering pass at different levels of abstraction for tile selection (Sect. 3.2) and ground texture rendering (Sect. 3.4). The third is a rather intricate module developed with CUDA for analysis purposes and frame-to-frame coherence (Sect. 3.3). We draw on the impressive computing power of the G80 GPU and some useful new properties: memory extended to 768 MB, texture size of up to $8192 \times 8192$ pixels and improved balance between vertex and pixel shader instruction processing.

From this point on, we suppose the entire terrain to be covered by a virtual static regular grid of tiles (RTG, Fig. 1). Among the important properties of our system, there are:

- **Negligible computation is performed on the CPU.**
- **It is a generic framework:**
  - it can be used with arbitrary ground meshes, generated by arbitrary algorithms, since all geometric input data are accessed only from vertex shaders
  - integration and performance tests are made easier.
- **All important computations usually performed by the CPU are transferred and processed on the GPU at low cost, and with simple shader instructions performing:**
  - texture frustum and occlusion culling (which could easily be extended to geometry data sets)
  - quick access to all geo-referenced data in the GPU shaders
  - simple and dynamical LOD computation per tile, without adjacent tile query. A speed control for the LOD is activated for high-speed navigation
  - virtual handling of hierarchical data structures in parallel with the corresponding structures stored on disk (e.g. quadtree structure, never stored in the GPU, but procedurally accessed).

- Frame-to-frame coherence of the streaming is done entirely on the GPU.
- Optimal reduction in the volume of data transferred between the CPU and the GPU, through the use of compression, Pixel Buffer Object (PBO) and CUDA programs.

The first module—TLM (Tile Load Map)—computes for the current viewpoint, in one render-to-2D texture pass, a visibility map for all of the terrain tiles. It deals with their frustum/occlusion culling and some associated level-of-detail values. This map is sent to the CUDA module, which computes the list of useful tiles within the quadtree. This list is then compared to the previous one, on the CPU, to maintain frame-to-frame coherence, resulting in the Ground_tiles Load Map (GLM): a short list containing the appropriate indices of the tiles that will be downloaded to the GPU memory. To handle large textured terrains, we use a simple but efficient out-of-core data process, with texture compression, to give with minimal latency, asynchronous transfer of the required tiles (given by the GLM list) from hard disk to the GPU memory into reserved “texture caches” in the GPU memory, which can be dynamic texture atlases or dynamic “texture arrays.” This point will be discussed in Sect. 3.4. During the second rendering pass, the ground mesh is sent to the GPU to be fully textured and shaded.

### 3 Implementation and technical details

#### 3.1 Required data

Our system is designed to work with fixed-time orthographic colored pictures of the Earth, and does not do any dynamical sun-shading. Original textures are packed into a classical Mitchell-filtered quadtree structure made up of $T^2$ pixels tiles. To avoid system latency during access, these are stored on disk, in directories containing only a few tiles, each one having its quadtree index as filename (see Sects. 3.3 and 3.4). We chose 512 as a value for $T$, to avoid fragmentation and give a good balance between transfer from storage media on PCI-16x, the number of OpenGL calls and a reasonable filling of the texture memory space on the GPU. In order to solve the continuity problem inherent in filtering in discontinuous texture spaces, a border size $b$ is assigned to each tile, as in [13], whatever its level within the quadtree. This border may be included in the tile, or placed outside, to keep the original database resolution. In this case, the tile size becomes $(T + b)^2$ pixels. Each tile has its own mipmap pyramid, compressed into a 1:6 lossy DXT1 format (to minimize texture-volume transfer and GPU decompression time).

For each frame, the ground geometry is computed by the “terrain mesh algorithm” (CPU, GPU, or mixed) and cached in a VBO during an initial “dummy rendering pass”
At this point, the depthMap texture (used in Sects. 3.2 and 4.4) is also stored in the GPU memory.

### 3.2 The “Tile Load Map” (first module)

This section deals with the core component of the tile-streaming-and-rendering architecture: the TLM algorithm (Fig. 2). This crucial module, which gives its name to the architecture, is dedicated to the selection of all the geo-referenced tiles required to display the current viewpoint. The computation is performed in one efficient render-to-texture pass, using a PBO and the subtle vertex (VS1) and fragment (FS1) programs, as described below. It produces the TLM: a texture of the same size as the RTG grid, which can be regarded as the first level of the tile-quadtree information.

**Shaders**

VS1 is mainly used to compute global texture, screen and windows coordinates. It also calculates the distance $D$ between the viewpoint and the current vertex, whereas FS1 is used to compute visibility and level-of-detail parameters, based on those varying values. It outputs a four-feature structure for each fragment: a visibility index ($v_{\text{index}}$), two LOD parameters (LOD$\_\text{min}$, LOD$\_\text{max}$) and a special one ($p_{\text{Index}}$) reserved for applications (see Sect. 4).

**Computed coordinates**

As mentioned in [26], the ground mesh is drawn in the virtual texture space corresponding to the entire terrain (Fig. 1), using normalized global texture coordinates UV within $[-1, 1]$. In terrain applications, the UV no longer need to be transmitted by the application, and this saves graphics bandwidth. They can be efficiently computed in vertex shader, on the basis of input world vertex coordinates and provided terrain size. Assuming the whole terrain to be quite flat, the projection method that we use in VS1 is 2D and linear. But it could be planispheric for wider and curved portions of the Earth and involve 3D computation. In VS1, the corresponding vertex screen coordinates (SC) and pixel window coordinates (WC) are also computed.
with the provided ViewProject matrix for the current viewpoint and window size.

**Frustum culling**  Because we only need to draw visible tiles in the TLM texture, a culling step is required, and it involves geometrical, non-commonplace computations (see Shader, below). This operation, usually devolved on a vertex shader, is performed in the fragment shader, for better precision and result stability, without a cost penalty (on the G80 only). In FS1, up to 6 clipping planes are used to frustum-cull the ground geometry polygons in texture space, in a predefined order that optimizes the computation cost. We implemented a distance-dependent frustum culling, where the “culling cone” is enlarged to avoid missing fragments around the viewpoint (the tuning of the algorithm and the parameters was seriously intricate and tedious!).

**Occlusion culling**  This mechanism avoids downloading an unnecessary number of tiles corresponding to hidden geometry for the current viewpoint. Our implementation is very simple, and does not need complex geometrical computation. We just compare the depth of the point in the world space coordinates associated with the current fragment to the depthMap of the scene. To do this, rasterized WC coordinates from VS1 are used in FS1 to compute a lookup in the high resolution depthMap texture previously rendered from the current viewpoint (see Sect. 3.1). A small empirical tolerance distance is added so that visible objects will not be lost during the depth comparison (see Fragment Shader FS1).

**Visibility index**  A v_index value of 1 is used to mark visible fragments in the current viewpoint, which are those that survive the previous culling tests. All culled fragments return a 0 value for v_index. The conservative rasterization and blending mode described in the “output” section is used to combine all the fragments for each tile and obtain an accurate “tile culling” method, so that a tile is marked as “visible” as soon as one of its parts is “visible”.

**Level of detail (LOD)**  The principle involved in the TLM process (see Sect. 3.3) requires the computation of two values (LOD_min, LOD_max), corresponding to the minimum and maximum values of the LOD for all input fragments of a tile, without adjacent tile query.

Different mechanisms can be used to estimate the required tile LOD in a shader. In order to minimize the number of tiles needed to display the current viewpoint, S. Lefebvre proposes an elegant solution that consists of a direct mipmaping estimation provided by the GPU [26]. For this purpose, a dedicated “LOD texture” is created and cached on the GPU. Its mipmap pyramid is filled with integer values representing rising levels, starting from 0 to the maximum value for LOD. In FS1, a simple access to this texture with WC coordinates gives the current LOD value for each fragment. Here, we have to take account of a corrective parameter related to screen resolution in order to produce good mipmap estimations in texture space. We have implemented and tested this method and found it rather unstable. Moreover, it is poorly suited to terrain needs, and therefore requires expensive loop computations in the third module of our architecture (see Sect. 3.4).

When using a classical radial LOD estimation, thresholds are based on a single parameter Rad_thres. We obtain better and stable results, with a light and elegant formula and minimal computing cost (see Shader, below). Moreover, we will demonstrate (in Sect. 4.4) that this radial choice is the best solution for terrain amplification with trees and forests, thus reinforcing its relevance.

**Speed influence**  By taking the navigation speed into account in the LOD evaluation, we can impose dynamic control over the stream. We reduce the amount of requested texels during high-speed traveling, thus making it more fluid. We just have to introduce speed as a uniform parameter within the FS1 and FS2 shaders. The following formula (1) shows how speed $S$ and distance $D$ can influence the LOD computation, given three hard-tuning parameters $(a, b, S_{\text{min}})$. But this is a first rough approximation of the general problem of linking local LOD computation with pixel speed displacement, which should require screen derivatives for the LOD.

where $S > S_{\text{min}}$

$$LOD = \min(LOD_{\text{max}}, \quad LOD + \max(0, a \cdot S - b \cdot \log_2(D)))$$

(1)

**Output**  Following these descriptions, the interesting question is: how do we finally get the features mentioned $(v_{\text{index}}, LOD_{\text{min}}, LOD_{\text{max}})$ for each tile? We have to remember that since we are working in texture space, FS1 computes the features for all fragments coming from the rasterization of the ground mesh using the same texture tile. So, in order to combine these data and get accurate values for $v_{\text{index}}, LOD_{\text{min}}$ and $LOD_{\text{max}},$ we just have to enable the OpenGL “max blending” mode (glBlendEquation(GL_MAX)) before the launch of the rendering process. LOD_min has to be replaced by 1-LOD_min in FS1, since the blending computes a maximum. Moreover, to avoid missing out fragments, we must use a conservative rasterization [26]. Hence we turn on the polygon antialiasing mode with the OpenGL instruction: glEnable(GL_POLYGON_SMOOTH_HINT).

Finally, in order to optimize transfers, the TLM texture is linked to a PBO, and we use the Nvidia-recommended 8-bit/component GL_BGRA texture format (see Sect. 3.3).
Vertex Shader VS1 (grey parts are intended for p_Index computation, for use in Section 4.4).

```c
const uniform float rangeTerrain;  // extent of the terrain (in meters)
const uniform vec2 windowSize;
uniform vec3 posViewer;
varying float D;
varying vec4 SC;
varying vec2 WC;
varying vec2 XZ;

void main( void )
{
  // Normalized global texture coordinates within [-1, 1]
  gl_Position.xy = ( gl_Vertex.xy / rangeTerrain ) * 2.0 - 1.0;

  // Screen coordinates (OpenGL ViewProject matrix transformation)
  SC = ftransform( );

  // Distance viewpoint -> vertex
  D = distance( gl_Vertex.xyz, posViewer );

  // Windows coordinates (obtained with a classical perspective division)
  WC = ( SC.xy / SC.w + 1.0 ) * windowSize;

  // 2D vertex coordinates
  XZ = gl_Vertex.z;
}
```

Fragment Shader FS1 (grey parts are intended for p_Index computation).

```c
uniform sampler2DRect depthMap;
const uniform float Rad_thres;
const uniform float P_COUNT;  // number of generic seed-patterns
const uniform float D_maxTree;  // max tree visibility
varying float D;
varying vec4 SC;
varying vec2 XZ;
varying vec2 WC;

void main( void )
{
  float p_Index = 0.0, D_min = 2200.0, cull_min, LodRad;

  // Distance-dependent frustum culling. The “culling cone” is enlarged to avoid missing fragments
  if ( D > D_min )
    cull_min = 0.0;
  else
    cull_min = -1024.0;  // empirical tuning
  if ( dot( SC, vec4( 0.0, 0.0, 1.0, 1.0 ) ) < cull_min ) discard;  // near
  if ( dot( SC, vec4( 1.0, 0.0, 0.0, 1.0 ) ) < cull_min ) discard;  // left
  if ( dot( SC, vec4( 1.0, 0.0, 0.0, 1.0 ) ) < cull_min ) discard;  // right
  if ( dot( SC, vec4( 0.0, -1.0, 0.0, 1.0 ) ) < cull_min ) discard;  // bottom
  if ( dot( SC, vec4( 0.0, 1.0, 0.0, 1.0 ) ) < cull_min ) discard;  // top
  if ( dot( SC, vec4( 0.0, 0.0, -1.0, 1.0 ) ) < cull_min ) discard;  // far

  // Depth parameters and occlusion culling (see OpenGL Reference Guides)
  float Near = 1.0, Far = 150000.0;  // current application parameters
  float a = Far / ( Far - Near ), b = Far * Near / ( Near - Far );
  float depth = b / ( texture2DRect( depthMap, WC ) * r - a );
  if ( SC.z > ( depth + 10.0 ) ) discard;  // added tolerance for better precision

  // Radial level of detail (unlimited), without speed correction
  if ( D <= Rad_thres ) LodRad = 0.0;
  else
    LodRad = ( 1.0 + log2( D / Rad_thres ) ) / 255.0;

  // Pseudo-random computation of a seed-pattern index
  if ( D <= D_maxTree )
    p_Index = ( 1.0 + floor( mod( 881 * abs( cos( XZ.x * XZ.y ) ), P_COUNT ) ) ) / 255.0;

  // 4-component output structure
  gl_FragColor = vec4( LodRad, 1.0 - LodRad, p_Index, 1.0 );
}
```
3.3 TLM analysis on the GPU (second module)

This module builds the list of tiles likely to be downloaded to the GPU (usually in one or more “Texture-caches,” depending on requirements—see [10] for example) with conservation of temporal coherence. It optimally ensures the nesting between useful tiles extracted from the TLM texture and the corresponding tiles in their respective quadtrees. For this purpose, an important rectangular 16-bit 2D-texture stored both in RAM and in GPU memory—TexQuadTree—manages the ‘Texture-cache’ occupancy associated with the indexed tiles on hard disk.

Previously computed on the CPU with important limitations [2], this module is now fully implemented on the GPU in a CUDA program. We might mention two major advantages: full TLM readback to CPU memory is no longer needed, and we can extend the TLM size to the maximum limit available on the GPU.

Our module comprises five steps (Fig. 2): PBO (transfer), TLM_Analysis (CUDA kernel), IsValid (CUDA kernel), Scan (CUDA procedure) and Compact (CUDA kernel).

PBO The frame-buffer content associated with the TLM texture (computed in Sect. 3.2) is first transferred from the rendering space to the CUDA space (Fig. 3), with a Pixel Buffer Object (PBO), using an OpenGL glReadPixels instruction and the GL_BGR texture format. The non-visibility between the two spaces makes it necessary to unregister the PBO ① before the transfer ②, and to register it after the transfer completion ③, so that its content is visible in CUDA space. Then the PBO is mapped on a CUDA variable ④ which becomes the first input of the TLM_Analysis kernel. This step is quite expensive (average 0.53 ms), but indispensable in the current state of the CUDA API [11].

TLM analysis The TLM analysis algorithm encodes the quadtree, in a certain way. It does not use tree structures but rather a combination of behavior and an indexing method following the quadtree logic. It is an iterative process that builds every useful level of the virtual quadtree, using a CUDA kernel (TLM_Analysis), as described in the ‘encoding scheme’ section. The LOD_min and LOD_max values that estimate the LOD range of a tile (at level 0) are used in conjunction with the v_index values to be propagated upward.

Encoding scheme Each iteration deals with quads of four neighboring tiles. For each level ‘i’, all quads can give birth to parent-tiles at level ‘i + 1’. We compute the LOD_min (resp. LOD_max) value for a parent-tile as the minimum (resp. the maximum) of the LOD_min (resp. LOD_max) values of its children.

The analysis starts by checking LOD_min ≤ i ≤ LOD_max (property A) for each active (non-culled) tile of the quads. If a quad passes this test, a global list cListGPU, initially empty, is updated to store its children (registered tiles). But there are two other rules illustrated by Fig. 4 for an 8 × 8 TLM example.

First rule. At level ‘i’, active tiles not already registered in a quad and all checking the property A should be registered in cListGPU at level ‘i’. In order to better understand, we draw in yellow in Fig. 4(a), (b), (c) the tiles that become registered for each level. The twelve yellow tiles at level 0 are all unregistered in their quads and check property A. At level 1, the four yellow tiles are registered since they satisfy property A and most other parts of their quads (in red) are already registered (as children). It is the same for the two yellow tiles in level 2, which both have (1, 2) as values for LodMin and LodMax: their neighbors in the same quad (in red) that have (0, 2) values were already registered as children. So it is sufficient that the two yellow tiles satisfy property A in order to be registered.

Second rule. If there exist at least two active and not already registered tiles within a quad, one satisfying property A and the other not, then all the tiles not already registered in this quad remain unregistered, and they will give birth to a parent-tile in the next level. This is illustrated, in Fig. 4, by the blue-circled tiles which satisfy property A for the current level, and their neighbors in the same quad not already registered (red-circled) which do not satisfy property A. In this way, a parent-tile is never stored in a list with one of its children.

CUDA implementation This section is a little technical, and can be skipped in a first reading. Given the encoding scheme that we have just explained, the most intuitive way to implement such a parallel process, while ensuring complete independence among the data (SIMD = single instruction multiple data), is to analyze each quad element of a level ‘i’ by means of a single thread, to produce a parent level ‘i + 1’. Thus the same thread writes the consolidation result (the parent) instead of the lower left element, in order to prepare the level ‘i + 1’ which will be analyzed (Fig. 5). The number of launched kernels is the same as that of levels of detail.

Because the threads of different blocks cannot communicate with each other, writing in a sequential list is almost impossible on a GeForce 8800 GTX card with CUDA 1.0. The problem is solved by using CUDA 1.1 atomic functions, but not on a GeForce 8800. Hence, the loop construction of the different quadtree levels is implemented in the Host level:

- The TLM_Analysis kernel can be run for each level with the exact number of threads that are needed.
- The size of the cListGPU list is equal to the virtual quadtree size required to preserve the interdependence of the threads.
- The termination of each level is guaranteed before the start of the next level.
- If the processing of different levels was implemented at
kernel level, it would mean that several blocks of threads remained inactive after the construction of the first level. This would increase the warp scheduling, which would become useless.

- Finally, the number of registers by thread in CUDA must not exceed 10 if we want to obtain an effective implementation [12]. This constraint is not guaranteed if we introduce this loop at kernel level.

At the end of this procedure, cListGPU not only contains the visible tile indices for the current viewpoint, but also a large majority of ‘zeros’ representing discarded tiles. To eliminate them, the compact operation described in [36] is applied (Fig. 6). In our CUDA implementation, it consists of three steps: flag generation by the ‘IsValid’ kernel, exclusive ‘Scan’ procedure, and a final shift by the ‘Compact’ kernel. Each thread deals with two elements in order to minimize
Fig. 5 TLM analysis. A thread (green) only scans its four neighbors (blue) and stores the results in the father-tile of the next level (red). For levels 1 and 2 only, Lod_min and Lod_max values from Fig. 4 are shown in grey.

Fig. 6 Compacting cListGPU in 3 steps

the number of blocks on hold and speed up the process (especially useful when the kernel contains simple, direct instructions [12]).

 IsValid  This kernel receives the cListGPU list as an input, and returns Valid, a list of the same size. Each thread demonstrates the following behavior: if the item is zero, it writes the value 0 in Valid, otherwise it writes the value 1.

 Scan  The Scan procedure comes with CUDA as a program type. It performs an exclusive scan of the Valid list, the result being stored in a new list of the same size: Scan. It makes a call to the recursive main function preScanArrayRecursive. The details of the procedure are outside the scope of this paper, but they can be found in the scanLargeArrays section of the reference manual of CUDA 1.0 [11]. The important point is that the Scan procedure prepares for each cListGPU[i] element the index Scan[i] to which it has to be shifted within the Compact kernel.

 Compact  This kernel takes cListGPU and ScannedValid as inputs, and another list of the same size, cListGPUout, as the output. Each thread running this kernel only shifts the two elements of cListGPU associated with it. So, a non-zero element cListGPU[i] is written in cListGPUout[index] with a value index equal to the contents of Scan[i].

 CPU tasks  Finally, we only need to read back to the CPU the non-zero values of the cListGPUout list, which have been compacted to the left of the list. Although there are not many of these (about a hundred values), this readback is performed at a non-negligible cost (Sect. 5; Table 1). However, this transfer is essential, because a current GPU cannot launch OpenGL instructions, including texture downloading. We maintain Texture-cache coherence by using a stack containing its free positions. So we never have to free the Texture-cache. Comparisons between the current cListGPUout and the previous one are performed on the CPU, and result in a very short list, GLM (Ground Load Map), which stores the indices of the tiles to be loaded up from the hard disk. Given that on the G80, 8192 x 8192 pixels are available for the Texture-cache, no LRU algorithm nor overflow man-
agement is necessary. This saves computing time. Meanwhile, the TexQuadTree texture is updated to be used in FS2.

3.4 Ground texture rendering (third module)

Texture-cache In the TLM implementation, the cache has not been designed to handle textures of different sizes, though it is used to store textures of different LOD. Indeed, all quadtree tiles are of the same size, but correspond to variable extents on the ground, according to their level in the quadtree. This is efficiently managed by FS2. So, the required $512 \times 512$ pixel ground tiles (whose indices are read from the GLM) are first downloaded into the Texture-cache, which manages a dynamic pool of up to 256 sub-textures on the G80 (Fig. 7). This is enough for most terrain applications. But with previous GPUs, whose texture size is up to $4096 \times 4096$ pixels, care must be taken not to overload the memory space.

The G80 has introduced the concept of “texture array,” which allows an indexed set of textures of the same size to be bound to a shader as a single object. The equivalent for older GPUs is the common “texture atlas,” especially cut into regular tiles for the needs of the terrain. The motivations usually invoked for the “texture array” technique are performance and simplicity. However, as far as the TLM framework is concerned, we have found that the use of “texture arrays” or atlases gives the same results. Both are easy to access in shaders, and the choice does not affect graphic performance, for three main reasons: (1) in both cases, only a single draw call is required to render the whole terrain mesh; (2) OpenGL implementation enables full support of compressed texture formats; and (3) filtering produces the same behavior (when we do not have to deal with “tiled texture” patterns).

Shaders A second draw order for the ground mesh (which is already stored in a VBO) is sent to the GPU with a single call. The vertex shader VS2 then computes the same variables as VS1 (including UV), and added fog and cloud coordinates. The following phase, which is considerably trickier, is carried out by the fragment shader FS2, which mainly draws on a “physical→to←virtual” address translator used for encoding/decoding coordinates in the quadtree structures.

Level of detail For the sake of consistency, and whatever the method used, the LOD computation has to be strictly identical to that performed by VS1 (Sect. 3.2).

Quadtree decoding Here, FS2 implements a virtual handling of the quadtree tile structure in parallel with the corresponding structure stored on disk. “Virtual handling” means that the tree-structure of our quadtree is never stored in the GPU memory, but is procedurally accessed. For this purpose, the TexQuadTree texture (see Sect. 3.3 and Fragment Shader FS2) is the only level of indirection that we use, resulting in high performance.

To texture the current fragment, UV should be processed, in theory, in a complete loop between LOD_min and
A GPU Tile-Load-Map architecture for terrain rendering: theory and applications

LOD_max with series of scaling and translating transformations. This loop is required when using the LOD estimation method described in [26]. But our solution, driven by the radial LOD behavior, is much simpler and more efficient in the following sense: the radial computation of the LOD in FS1 gives a maximum amplitude of 2 for (LOD_max–LOD_min), so that a costly loop is replaced by at most three calls (a), (b) or (c) to the CheckQuad procedure, which involves one access to the TexQuadTree texture to detect the relevant LOD value. These calls are ordered, given the nature of the quadtree reconstruction performed by the CUDA step. We first test the current LOD value (58% successful) and, if rejected by the CheckQuad procedure, we test the LOD+1 value (40% successful). The LOD−1 test is successful for only 4% of the fragments. This gives an average number of CheckQuad calls equal to 1.44, which contributes to the overall performance. CheckQuad computes the tile index in the quadtree and the corresponding relative coordinates ‘uvT’ in this tile. An easy arithmetical calculus then gives the tile location (q0, r0) in Texture-cache. Finally, the computation of texture coordinates ‘UVC’ in the cache is immediate, and the fragment color can be read, filtered and shaded (see Shader FS2 below).

Filter control Our solution to solve the discontinuity problem that occurs when a mipmap filter is applied in this cache was explained in Sect. 3.1. The result is that any number of mipmap levels can be supported by our framework. However, given our LOD control and activation distances, it appears that it can be reduced to 3 or 4 levels without loss of visual quality, thus lowering bandwidth occupation. Free anisotropic filtering is also enabled in order to attain the best results in conjunction with mipmapping.

Atmospheric effects Based on dynamic time-morphing textures, clouds are projected at pixel level, which enhances the appearance of the ground in real time and gives more realism to the scene. Speed and direction control is integrated. An added exponential fog is tuned so that distant mountains appear blue and blurry, as in nature before noon.

**Vertex Shader VS2.**

```cpp
const uniform float rangeTerrain;
uniform vec3 posViewer; // Dynamic cloud parameters
uniform vec4 cloudParam; varying vec2 UV;
varying vec4 UYCloud;
varying float fogZ;
varying float D;

void main(void)
{
    // Screen coordinates (OpenGL ViewProject matrix transformation)
    gl_Position = ftransform();

    // Global texture coordinates
    UV = gl_Vertex.xyxz / rangeTerrain;

    // Cloud and fog
    UYCloud = cloudParam.xxyz * gl_Vertex.xxyz + cloudParam.zxyz;
    fogZ = gl_Position.z;

    // Distance viewpoint -> vertex
    D = distance (gl_Vertex.xyz, posViewer);
}
```

**Fragment Shader FS2.**

```cpp
uniform sampler2D TexCache; // Texture-cache
uniform sampler2DRect TexQuadTree; // Texture coding tile locations within TexCache
uniform sampler2D TexCloud; // Cloud Texture
const uniform float Rad_thres;
const uniform vec2 decal; // for tile border handle
uniform float ResolTile; // Tile resolution (in pixels)
uniform float ResolCacheTile; // Cache resolution (in tiles)
uniform float ResolCachePixel; // Cache resolution (in pixels)
varying vec2 UV;
varying vec4 UYCloud;
varying float fogZ;
varying float D;

// Efficient computation of the tile position in TexQuadTree and the relative coordinate uvT within this tile
float checkQuad (float lod, vec2 uv, out vec2 uvT)
{
    float r0, q0, DD, R, Id, RR = exp2 (floor (lod));
    R = Resol_Niw_0 / RR;
    vec2 quot = floor (R * uv);
    uvT = (R * uv - quot) * ResolTile;
    DD = ((RR * RR - 1.0) / 3) * 4 * R * R;
    Id = DD + R * quot.y + quot.x;
    q0 = floor (Id / ResolTile);
    r0 = Id - q0 * ResolTile;
    return (texture2DRect (TexQuadTree, vec2 (r0, q0)) .x);
}
```
4 Application to terrain amplification

During real-time terrain navigation, altimetric and photographic database resolution inevitably becomes insufficient when the viewpoint reaches ground level. While limiting memory load and computing costs, it is desirable to add, as soon as necessary, geometrical details for the mesh and textural details for the ground textures, and to allow the emergence of sets of plant and mineral objects adapted to the viewpoint requirements. Obtaining such an enriched reality from restricted data sets is known as “amplification.” The term was first mentioned in \[40\].

This feature is required in many applications dealing with terrain rendering. But current earth navigators—such as Google Earth—are designed to stream and display digital models of real textured terrains without amplification, whereas Microsoft “Flight Simulator,” without providing access to real aerial textures, implements a kind of amplification by mapping pre-equipped generic patterns onto the ground. A first attempt to reconcile the two approaches was made in 2001 by the French company EMG via Eingana \[20\]. Based on limited-resolution databases and using various analytical tools, Eingana generated a fairly realistic 3D planet by fractal amplification.

4.1 TLM suitable for amplification

Nowadays, with the increasing power of graphics hardware, injecting texture synthesis procedures into GPU shaders—which was a serious challenge just four years ago—has become affordable, and we can apply smoothly-controlled and on-the-fly amplification techniques to natural scenes, especially for ground (fine-scale mesh and texture). Given that TLM deals with arbitrary ground meshes, geo-referenced vertices can be added to them easily and independently on the G80 by the use of Geometry Shaders. For better realism, added features must be linked in real time to external ground texture analysis and geo-morphological databases. Some promising work has recently been done in this respect \[10, 13, 27, 38\].

The design of the TLM architecture, by its nature, facilitates the integration of geometric input-data into procedural selection and rendering tasks on the GPU. It allows “on-the-fly ground texture amplification” to be carried out easily, in conjunction with seasonal behavior (for example, covering ground tiles and plants with snow). It can draw on a geotypical set of detailed materials that can be geospecifically applied and gradually faded at a distance with the native ground textures. Some material-typical transition functions are required to avoid repetition artifacts and ensure continuous, detailed, synthesized scenes. Researchers generally use cached procedural textures, or on-the-fly computation, which is easier to integrate at fragment level within the second TLM pass.

But this architecture is also perfectly suited to the extraction and visualization of natural objects on the ground, meeting qualitative and performance criteria. Here, some components of the TLM are used to calculate and display quite realistic procedural landscapes and ecosystems with minimal pre-stored data, CPU load and popping artifacts. The result is a full GPU seed model designed for on-the-fly plant (or mineral) amplification, working in tandem with a pre-classification of available aerial images and making it possible to fine-tune the properties of objects added to the scene.
4.2 Implementing plant amplification

There have been a large number of studies on the production of plant distributions, and in particular, procedural instantiation, which avoids explicit tree locations. In this context, seed models are extremely useful in the specification of landscapes. Exploiting the high redundancy that is present in nature, they make it possible to populate terrain on-the-fly without storing all the different objects and their properties, while preserving the variety of the results.

**Seeds and patterns** Visible objects in aerial textures (trees, shrubs, rocks) are well described by seeds, making it easy to take into account geographical (altitude), climatic (moisture), seasonal (snow in winter) and temporal (growth, ageing) rules, using a parameterized model. Seed-patterns are used to store such distributions, to take account of local species density and to compose virtual patchy landscapes. Well described in [17, 18, 25, 26], they usually consist of squared generic sets of seeds or seed clusters. They can store one or more species, which makes them useful as a way of simulating varied and adaptable ecosystems. Random or controlled variations can be introduced on this level to avoid strict repetition of generic fittings. A first GPU implementation of this technique to generate random forests can be found in [26], whereas [43] describes how to use terrain slope and land cover classification (LCC) to distribute vegetation. [8] shows how to use land usage databases to build geo-specific landscape content.

**On-the-fly GPU seed model** The on-the-fly amplification concept does not allow for the pre-storage of seeds [44]. We therefore developed an adaptive seed model—inspired by previous publications—in which seed placement and color are constrained by the content of the aerial images, whose visible detail must be finely restored. Our model is fully implemented in two dedicated GPU modules for “seed selection” and “seed rendering.”

Since there is no existing real-time method for computing “land cover classification” (LCC) for each pixel, we use a pre-classification step for aerial images. This is the current trend [9, 34, 38], and it draws on recent cooperation with INRIA/MISTIS [6, 7] and IGN/MATIS [42] using statistical modeling applied to the detailed analysis of colored images. We demonstrate in the next sections how to use LCC values (in a density-map-like approach) to extract seeds on-the-fly and produce a plausible scene.

**Real-time plant rendering** Some of the tools—e.g. [1, 30]—that are used to generate realistic 3D plant models are unsuited to real-time rendering of dense scenes with thousands of trees. Alternative representations are more efficient, and maintain good visual quality, e.g. volumetric textures [15, 32], image-based or point-based approximations [14, 22], or a combination of geometry-based and volume-based approaches [9]. It is often difficult to attain the level of individual objects, which rules these methods out for seed rendering in the context of real-time amplification. Simple or crossed billboarding has been widely used for plants in real-time applications, but these methods lack relief and parallax, even if view-dependent textures are used. Billboard clouds, by creating volume, improve realism, but not thickness. In [16], X. Decoret proposes a simplified method for generating sets of static billboards that approximate plant geometry, provide good parallax and give an impression of depth. And A. Fuhrmann has improved his algorithm to cope better with trees [21]. We used this technique in our experiments (Sect. 6).

4.3 Packing LCC values in a quadtree structure

We store LCC values in textures, on disk, for access in a quadtree, since we have to retrieve them at different levels of detail (trees being visible up to 3,000 m, their LCC values are spread over multiple levels in the quadtree). The coding scheme shown in Fig. 8 is not commonplace, because ground textures mapped to the terrain are not necessarily of the highest resolution at which we have to locate the LCC values. All these constraints impose a redundant coding that should be consistent between all the relevant levels, if popping objects are to be avoided. Moreover, we cannot pack ground colors and LCC values together in the same 32-bit RGBA tiles, because the latter must not be compressed.

In sum, a specific “LCC quadtree” is needed to pack LCC values into luminance tiles. We use 3 levels for trees. To ensure consistency between these different levels, we propose the following coding method, which allows a 1 m positioning precision on the ground (which is sufficient for mountains). Level 1 uses 8 bits to store each LCC type resulting from the classification. We duplicate these values on 4 neighbors at level 0. Lastly, level 2 uses 8 bits to encode 4 neighboring texels from level 1, meaning that 3 different species can be stored on 2 bits (0 is kept for bare ground). We can increase the coding precision and the number of species supported by allocating more bits to the LCC texture format. With 16 bits, 15 species can be addressed, and with 32 bits, 255 species, although this means using more GPU memory and bus bandwidth.

![Fig. 8 LCC type quadtree-packing diagram (A, R and N are examples of LCC values)](Image)

Springer
4.4 Visible seed selection (pass 3)

Upon the launch of the application, a set of $P$ generic square patterns containing random virtual seeds is produced (only their 2D positions are computed) and stored in the GPU memory. We design patterns whose number of seeds approximates the plant or mineral density for a typical zone. Plant and mineral objects are also cached in the GPU memory.

To retrieve visible seeds with minimal computing time, only visible patterns (in a radius corresponding to the visibility range of a tree—usually 4000 m) are extracted and stored in a Pattern Load Map (PLM). For this purpose, we slightly complete the TLM pass 1 to compute an aperiodic, stable set of seed-pattern indices (see the grey part of Fig. 2: Seed Pattern Instancing). Since all ground tiles, LCC tiles and seed-patterns are designed to share the same extent, FS1 computes all these values together for each tile output structure, which is a judicious advantage.

Once the PLM is available, in the same way as for the tiles the graphic pipeline is programmed for a “one-pass rendering” in a 2D texture called SLM (Seed Load Map). The size of this texture is at most $512 \times 512$ pixels, which can store enough seeds to populate the front view of large scenes. All the seeds are processed by the graphics card and sent from the PLM by OpenGL calls to the patterns, which are 1D arrays initially cached on the GPU (Fig. 9) in vertex buffer objects (VBO). Each seed structure stores 8 GPU-computable properties: 3D position on the ground, LCC type (species), surrounding sphere (center, radius), level of detail, size, orientation, ground color and an “index” within the VBO. The surrounding “sphere” qualifies the seed instance, whose center is placed at mid-height from it. The “ground color” is designed for a colorimetric follow-up of the terrain (end of Sects. 4.4 and 4.5). Pseudo-random functions are used to disturb size, orientation and color, in order to obtain non-redundant variations, as in nature. Except for the “index” (which is used to compute the output position in the SLM), scalar heights are transmitted to the output structure of the fragment program FS3, for use in pass 4. But a texture can store up to 4 floating components. So the ARB_drawbuffers extension is enabled in order to activate the Multi Render Target (MRT). FS3 returns a special value if the corresponding seed is invalid, hidden, or outside the viewing frustum. The five following steps describe in detail how seeds are selected and stored in the SLM.

**Pattern_index** In the TLM pass (FS1), the reserved fourth component $p_{\text{Index}}$ (described at the beginning of Sect. 3.2) is now used to store up to 255 possible seed-pattern indices, which are computed on-the-fly for each frame, de-
pending only on a germ choice. This avoids storing memory-consuming arrays in the CPU or GPU for large terrains and allows some geographical, seasonal and temporal rules to be taken into account for optimal placement in VS1. Up to now, we have used a space-time-coherent, aperiodic, pseudo-random generator which assigns to each tile a pattern index \( p_n \in [1, P] \). At the end of this pass, the required seed-patterns are grouped by type, so as to optimize the number of VBO calls in pass 3.

Seed positioning on the ground  Generic patterns only store seed locations \((x, y)\), therefore seeds are first positioned in 2D, using translations performed in VS3. This is followed by an orthogonal projection onto the ground mesh to compute the \( z \) value, given that the heightmap texture is pre-loaded onto the GPU. Since the ground mesh is processed by a multiresolution algorithm, values for \( z \) may be view-dependent, except when the viewpoint is close to the ground. To avoid visual artifacts, all values of \( z \) must be the same, whatever the level of detail of the ground. We decided to anchor objects to the ground on their smallest \( z \) component, obtained by a bilinear interpolation on four \( z \) neighboring values. Lastly, we ensure that the lower part of the trunk goes down a little way into the ground. This technique is particularly suited to amplified terrain meshes with detailed heightmaps whose amplitude is known in advance.

Seed visibility  This takes place in two steps, like the TLM process. VS3 performs a culling test: a seed whose surrounding sphere is totally exterior to the frustum is removed. Let \( R \) be the transformed radius of the sphere (taking account of on-the-fly random “size” computations) and \( \epsilon \) a tolerance value. We project the center of the sphere into the “Normalized Device Coordinate” (NDC) space, then check if it belongs to \([-1 - R - \epsilon, 1 + R + \epsilon] \times [-1 - R - \epsilon, 1 + R + \epsilon] \times [0, +\infty[\). The occlusion-culling test is performed in FS3 and discards a seed (hidden by the ground) if its depth is greater than the value stored in the depthMap. For maximal precision, this value is computed at the “summit” position of the object projected onto the screen. For distant seeds, it is necessary to take account of the mesh simplification performed by the terrain multiresolution algorithm. An empirical tolerance distance is added so that visible objects will not be lost. Lastly, we enable the GL_CLAMP for the nearest mode of the depthMap texture, given that an object whose summit does not appear on the screen may not have an associated screen depth.

LOD computation  This is performed in vertex shader VS3, which outputs a radial LOD evaluation for each seed combined with its pseudo-random size, so as to take into account the visual print during the LOD transitions. When using the speed correction (Sect. 3.2), we clamp the modified LOD value in order to maintain tree visibility, except for very high speeds, for which drawing trees becomes useless.

The power of the TLM procedures is then used to read LCC and ground color values from their respective texture space in the GPU memory:

LCC reading  The virtual seed species is instanced in FS3 for each seed: we access values in the LCC quadtree using the encoding scheme and the very efficient methods set out in Sects. 3.2 and 3.4. FS3 discards the current seed if no valid LCC type is found.

Here we can justify again the radial choice made for the TLM computation of the LOD component of the TLM texture. As regards LCC tiles and nested seed-patterns, we cannot use the mipmapping mechanism proposed in [26] to estimate the LOD, because a user on the ground or at a peak would be likely to obtain horizontal polygons whose high-mipmapping LOD values would result in erroneous LCC readings and prevent the display of trees. To eliminate this problem, the distance \( D \) has to be used to evaluate a radial LOD. When a seed-pattern is visible at a distance of less than 4,000 m, a radial selection is computed for the first 3 LODs of the LCC quadtree \((0, 1 \text{ or } 2)\), values of 3 or more being reserved for non-active patterns.

Ground color reading  This is accessed in FS3 using the same efficient method given for ground-tile rendering in Sect. 3.4. A random sampling of ground colors is carried out in the neighborhood of each seed. This allows fine modulations of plant or mineral colors to take place in pass 4.

4.5 Visible object rendering (pass 4)

The choice of the rendering algorithms is independent of the previous step, but five parameters (stored in the SLM) ensure their nesting: species, size, orientation, LOD and ground color. On the G80, we render selected entities with multiple hardware instancing calls, while avoiding a complete SLM readback in the CPU, because instancing properties can be accessed in shader with SLM texture readings. Given \( M \) mineral and \( V \) plant objects, expressed in \( N_M \) and \( N_V \) LODs, the number of instancing calls is \( (M.N_M + V.N_V) \). To specify the “instancing stream” of each call, we just have to know the number of primitives to send. Since the SLM contains not only “visible seeds” but also more numerous “holes” (corresponding to discarded seeds), we use a GPU “stream reduction” algorithm. This has several advantages: filtering capabilities that sort seeds by type, pack them without holes and access to the number of packed seeds, which is transmitted to the CPU at negligible cost using a readback of very few pixels. We implemented the process described in [36], with the help of its author.
The colorimetric ground follow-up naturally takes account of shadowed zones, and reduces unnecessary computations. The colorimetric transition is performed by a smooth blending with the ground textures at the rendering step of each instance. The foliage and trunks of the trees are processed independently, and the luminosity of the textures is modulated according to their relative positions within the object. As for ground rendering, dynamic time-morphing clouds are projected at pixel level onto these objects, in the interest of realism. All shading is performed in FS4.

5 Experiments and results

For best results, the TLM architecture requires a compatible “shader model 4 GPU” with at least 512 MB of RAM. But it can run on older GPUs, without using texture arrays and instancing extensions. All the codes were developed in C++, OpenGL, GLSL and CUDA. The following tests were performed on a dual-core computer running at 2.93 GHz, with 2 GB RAM and a GeForce 8800GTX. The Nvidia graphic driver 162.01 was chosen due to the fact that it works well with CUDA 1.0.

For our experiments, we used databases from the Haute-Savoie region: an area of 4,388 km² in the French Alps. The terrain used for demonstration covers a 4097 × 4097 DEM (digital elevation model) at 16 m/vertex resolution, equipped with a set of 0.5 m/pixel aerial photographs. All databases were provided by [35].

We processed a 256 × 256 pixel TLM and associated 512 × 512 pixel ground/LCC tiles. We did not use any mineral objects for the amplification, though it is possible, and relatively easy, to do so. Only trees were used for the experiments, precisely because they constitute the most difficult challenge. The maximum forest density is 21,000 trees/km², and the height of the species ranges from 2 to 25 meters. We used rather realistic billboard cloud models of typical trees [21], with three LODs composed respectively of 17, 8 and 4 quad-billboards. Each one was textured with a single 16-bit luminance atlas image of at most 4096 × 2048 pixels.

For demonstration purposes, RMK2 [4], based on an improved version of the SOAR algorithm [29], was the library chosen to carry out the ground-mesh rendering. An improved implementation of the TLM architecture using geometry clipmaps [3] is in progress and will provide higher frame rates in terrain rendering. But whatever the method used for constructing the terrain mesh, the following results remain valid.

5.1 Qualitative results

Currently, our system supports amplification and display of terrains up to 2097 × 2097 km² at 0.5 m per pixel resolution. On one hand, this limit is related to the current texture size available on the G80. On the other hand, it is due to the current TLM mechanism (although developed from an efficiency point of view), which projects all visible ground vertices in texture space.

The navigation is fluid and contains no popping effects that could be due to aggressive LOD or rough simplifications. This is an advantage over clipmap methods, which use a type of texture interpolation that leads to visual morphing artifacts. Moreover, the use of SOAR for the mesh has provided total control over visual error, thus ensuring navigation without disturbance.

In comparison with Google Earth, our rendering engine maintains imperceptible transitions between successive levels of detail for mesh and texture. And it automatically adds, in the same way, tens of thousands of trees per frame in areas where they are visible on aerial photographs (Fig. 11).

5.2 Performance analysis

Computing times for the all selection/rendering tasks are shown in Table 1, for terrain rendering only, and in Table 3 with added vegetal amplification. They give an overall impression of the performance of our system (Fig. 12), especially since the average CPU charge devoted to our architecture is quite small (<15%). Table 1 and Fig. 12 give an optimized, steady result for the TLM engine, which takes between 1.5 ms/frame and 2.0 ms/frame. It is important to note that the shader computing time seems negligible in comparison with the time reserved for obligatory transfer operations (PBO, readback, streaming). Indeed, the three main passes (3.2–3.3–3.4) which appear at the bottom of the graph sheet are very efficient.

Surprisingly, the pass 1 computing time (Table 2) does not depend on the size of the TLM, and is quite stable between 0.06 ms and 0.07 ms). We obtain the same result for pass 2, but this is obvious. Indeed, only the analysis pass depends strongly on this size and reveals quadratic behavior (see Table 2).

The influence of resolution is obvious for fragment passes (especially pass 4: see Table 3). However, pass 2 remains the least time-consuming part of our rendering architecture. Computing time for pass 3 depends strongly on the number of checked seeds per frame (which is related to the overall plant density), whereas the SLM packing process is more or less dependent on this number. We might also note that the results are not influenced by the number of generic seed-patterns used. The ratio between, on the one hand, the selection/generation time of the models and, on the other hand, the rendering time, may be modified or even improved by implementing methods other than that of cloud-billboards.

Comparisons with CPU techniques show that the TLM and SLM algorithms on the GPU are between 2.5 and 10
times faster than their CPU implementation. Frustum and occlusion culling substantially reduces the computing costs, though much more on the GPU than on the CPU, and saves bandwidth. When the TLM size is below $256 \times 256$, the CPU implementation is faster. But beyond this value, CPU computing time far exceeds that of the GPU, because the full TLM readback, which is indispensable, becomes a bottleneck. And the TLM analysis with a parallel architecture becomes very efficient for high size values.

In pass 2, the transition between levels 0 and 1 of the ground quadtree is tuned to eliminate visual popping, with optimal $\text{Rad}_\text{thres}$ value equal to 1200 m. With this setting, 3 to 4 mipmap levels per tile only, and the $4 \times$ anisotropic filter activated, we obtain good display quality (Figs. 10, 11)
Table 1 Performance analysis (in ms) of the GPU passes for terrain only, measured for three typical viewpoints and two common screen resolutions 1280 × 1024 (1) and 1024 × 768 (2)

<table>
<thead>
<tr>
<th>Viewpoint</th>
<th>Resolution</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build mesh (CPU)</td>
<td></td>
<td>5.63</td>
<td>4.4</td>
<td>5.41</td>
<td>3.882</td>
<td>1.5425</td>
<td>1.374</td>
</tr>
<tr>
<td>Mesh VBO and depthMap</td>
<td></td>
<td>1.271</td>
<td>1.268</td>
<td>1.288</td>
<td>1.265</td>
<td>1.262</td>
<td>1.257</td>
</tr>
<tr>
<td>Pass 1</td>
<td></td>
<td>0.0641</td>
<td>0.06</td>
<td>0.066</td>
<td>0.061</td>
<td>0.0633</td>
<td>0.0618</td>
</tr>
<tr>
<td>PBO transfer for CUDA</td>
<td></td>
<td>0.558</td>
<td>0.557</td>
<td>0.556</td>
<td>0.546</td>
<td>0.511</td>
<td>0.546</td>
</tr>
<tr>
<td>TLM analysis with CUDA</td>
<td></td>
<td>0.1191</td>
<td>0.1188</td>
<td>0.1193</td>
<td>0.1191</td>
<td>0.1167</td>
<td>0.1166</td>
</tr>
<tr>
<td>GLM readback</td>
<td></td>
<td>0.421</td>
<td>0.42</td>
<td>0.419</td>
<td>0.418</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>Tile streaming tasks</td>
<td></td>
<td>0.751</td>
<td>0.548</td>
<td>0.802</td>
<td>0.721</td>
<td>0.554</td>
<td>0.378</td>
</tr>
<tr>
<td>Pass 2</td>
<td></td>
<td>0.034</td>
<td>0.03</td>
<td>0.033</td>
<td>0.032</td>
<td>0.0331</td>
<td>0.03</td>
</tr>
<tr>
<td>TLM_only</td>
<td></td>
<td>1.947</td>
<td>1.734</td>
<td>1.995</td>
<td>1.897</td>
<td>1.699</td>
<td>1.553</td>
</tr>
<tr>
<td>Terrain_rendering_FPS (Hz)</td>
<td></td>
<td>113</td>
<td>135</td>
<td>115</td>
<td>142</td>
<td>222</td>
<td>239</td>
</tr>
</tbody>
</table>

Table 2 Performance (in ms) of TLM pass 1, and analysis with CUDA, for different resolutions

<table>
<thead>
<tr>
<th>TLM size</th>
<th>128 × 128</th>
<th>256 × 256</th>
<th>512 × 512</th>
<th>1024 × 1024</th>
<th>2048 × 2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 1 (ms)</td>
<td>0.066</td>
<td>0.07</td>
<td>0.071</td>
<td>0.073</td>
<td>0.081</td>
</tr>
<tr>
<td>Analysis (ms)</td>
<td>0.114</td>
<td>0.1183</td>
<td>0.5374</td>
<td>2.065</td>
<td>6.467</td>
</tr>
</tbody>
</table>

Table 3 Performance analysis (in ms) of a tree-amplified terrain, with a large number of checked seeds per frame, measured for three typical viewpoints and two common screen resolutions 1280 × 1024 (1) and 1024 × 768 (2)

<table>
<thead>
<tr>
<th>Viewpoint</th>
<th>Resolution</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 3</td>
<td></td>
<td>0.683</td>
<td>0.661</td>
<td>0.317</td>
<td>0.303</td>
<td>0.221</td>
<td>0.212</td>
</tr>
<tr>
<td>SLM packing</td>
<td></td>
<td>0.531</td>
<td>0.539</td>
<td>0.507</td>
<td>0.510</td>
<td>0.379</td>
<td>0.375</td>
</tr>
<tr>
<td>SLM readback</td>
<td></td>
<td>0.493</td>
<td>0.494</td>
<td>0.497</td>
<td>0.493</td>
<td>0.495</td>
<td>0.492</td>
</tr>
<tr>
<td>Pass 4</td>
<td></td>
<td>8.405</td>
<td>4.701</td>
<td>5.863</td>
<td>3.931</td>
<td>2.382</td>
<td>1.331</td>
</tr>
<tr>
<td>Selected seeds</td>
<td></td>
<td>72769 / frame</td>
<td>32807 / frame</td>
<td>13687 / frame</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checked seeds</td>
<td></td>
<td>233920 / frame</td>
<td>141 728 / frame</td>
<td>115 712 / frame</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total_FPS (Hz)</td>
<td></td>
<td>50</td>
<td>71</td>
<td>62</td>
<td>80</td>
<td>121</td>
<td>127</td>
</tr>
</tbody>
</table>

at low cost, without overloading the graphics bandwidth, especially if the speed correction is enabled.

Finally, Table 1 shows that the global fps is highly influenced by the poor performance of the SOAR CPU-algorithm. But with GPU clipmap implementation, this fps is about to reach 200–250 Hz for most viewpoints.

6 Conclusions and prospects

We are putting forward a robust, comprehensive GPU approach for the real-time visualization of vast terrains. For each frame, the selection of required textures from quadtrees and seeds is performed entirely on the GPU, driven by an optimal combination of two visibility algorithms. Our architecture, by its nature, allows on-the-fly ground amplification and population with tens of thousands of natural elements, by the use of powerful streaming and instancing algorithms. The current implementation has been carefully and highly optimized on the G80. But it remains open to future GPU improvements in the handling of terrain meshes and added objects.

Our current objective is to extend the TLM algorithms to the rendering of entire (and entirely textured) planets, at meter resolution. One possible approach has been given in this paper, via CUDA, to benefit from the size of the TLM texture. But it is not enough to deal with a country whose size is larger than that of Brazil, and the computing time of the TLM analysis becomes a major weakness since the TLM texture is of the same extent as the terrain. We have to go further if we want to display the entire world using this mechanism. We believe an adaptive grid structure with variable extent would be better. We are starting with a rough 3D model of the earth in order to compute a first visibility map with an adaptive-TLM process, including only frustum culling. The dynamic downloading of texture and geometry data sets (even with details for amplification) provides everything necessary to launch a second TLM pass and a final rendering pass. On the other hand, implementation will
require the availability of databases on a large scale, or a simulation.

Acknowledgements This work was supported by the DEREVE II project in France from 2005 to 2007. Our deep thanks go to “the father of TLM,” Sylvain Lefebvre, for the time he spent explaining his original architecture to us, and helping us in our implementation. Our special thanks go to David Roger (for the adaptation of his “stream reduction” to the SLM simplification), Juliette Blanchet and Florence Forbes (for these last two years’ cooperation in applying Markovian analysis to our work), Arnaud Le-Bris and Gilles Martinoty (for the classification tests in the region of Megève) and Cyril Crassin (for his answers to our numerous technical questions about the G80 over the last year).

And of course we must not forget Sylvain Meunier, who worked with us on the first GPU seed model implementation in 2006, Peter Lindstrom and Andras Balogh (for advice on SOAR and RMK2 use), Anton Fuhrmann (for the billboard-clouds tree models) and RGD73-74 (www.rgd73-74.fr), our French partner and database supplier.

References

20. Eingana: Le premier atlas vivant en 3D et images satellite, Cdrom, EMG (2001)
42. Trias-Sanz, R., Boldo, D.: A high-reliability, high-resolution method for land cover classification into forest and non-forest. In: 14th Conf. on Image Analysis, Finland (2005)
Yacine Amara is a Ph.D. student in computer science. His field of interest deals with visualization and amplification of vast terrains. He is a lecturer in computer graphics, numerical computation, graph theory and optimization on networks, stochastic process, multivariate analysis and object-oriented programming at the Military Polytechnic School of Algiers.

Xavier Marsault is head researcher at MAP-ARIA lab in the Architecture School of Lyon. His research interests over the past decade include architecture/urban/landscape 3D-modelization, restitution and simulation, non-photorealistic rendering, pattern recognition, data compression and cryptography. He received in 1998 a Ph.D. in computer science from the Jean Monnet University of Saint Etienne in France for a thesis on applications to "Thom’s catastrophe theory" to image processing and analysis.