HAL
archives-ouvertes.fr

Real-time rendering and editing of vector-based terrains

Eric Bruneton, Fabrice Neyret

▶ To cite this version:

Eric Bruneton, Fabrice Neyret. Real-time rendering and editing of vector-based terrains. Computer Graphics Forum, Wiley, 2008, Special Issue: Eurographics '08, 27 (2), pp.311-320. 10.1111/j.1467-8659.2008.01128.x . inria-00207679

HAL Id: inria-00207679
https://hal.inria.fr/inria-00207679

Submitted on 18 Jan 2008

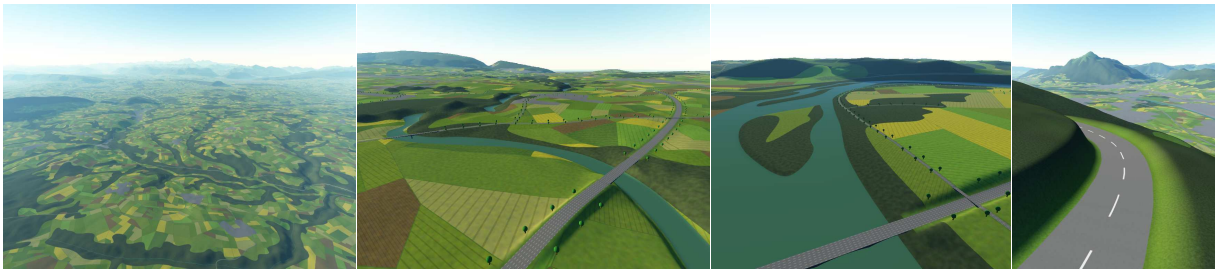
HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

EUROGRAPHICS 2008 / G. Drettakis and R. Scopigno
(Guest Editors)

Volume 27 (2008), Number 2

Real-time rendering and editing of vector-based terrains



Eric Bruneton and Fabrice Neyret

EVASION – LJK / Grenoble Universités – INRIA

inria-00207679, version 1 - 18 Jan 2008

Abstract

We present a method to populate very large terrains with very detailed features such as roads, rivers, lakes and fields. These features can be interactively edited, and the landscape can be explored in real time at any altitude from flight view to car view. We use vector descriptions of linear and areal features, with associated shaders to specify their appearance (terrain color and material), their footprint (effect on terrain shape), and their associated objects (bridges, hedges, etc.).

In order to encompass both very large terrains and very fine details we rely on a view dependent quadtree refinement scheme. New quads are generated when needed and cached on the GPU. For each quad we produce on the GPU an appearance texture, a footprint texture, and some object meshes, based on the features vector description and their associated shaders. Adaptive refinement, procedural vector features and a mipmap pyramid provide three LOD mechanisms for small, medium and large scale quads. Our results and attached video show high performance with high visual quality.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

Landscapes usually consist of precise *features* lying on top of a very large terrain. Features such as rivers, roads, lakes and fields could be considered as part of the ground. However simply including them in the raw terrain data gives imprecise results, even at very high resolutions (see Figure 1), because they have very precise shapes and boundaries, and often locally constrain the terrain shape (roads and lakes must be flat, rivers cannot flow uphill, etc.). Several applications such as Geographic Information Systems (GIS) provide them as vector data in separate layers. Hence the idea to combine a digital elevation model (DEM) with vector data to get precise feature boundaries and to enforce the previous constraints (with the additional possibility to cover the terrain with *objects* that are not part of the ground, such as bridges or hedges, generated from the same vector data). The need for interactive editing tools then comes naturally, to fix inconsistencies between data sources, to perform environmental impact studies, or for world editors for games.

In this paper we propose a GPU-friendly algorithm for the real-time quality rendering of such scenes, allowing their real-time editing. Our method combines terrain elevation,

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.
Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

appearance and precise vector data such as rivers and roads into a common data structure, based on a quadtree. Each quad holds the local information about the terrain: elevation, appearance and vector shapes, the latter being resolution independent. The elevation and appearance of each quad are cached in textures on GPU and are produced on the fly during exploration, by rasterizing the vector data at the appropriate resolution. This allows real-time rendering and editing of high resolution terrains. Rasterizing the vector data into the elevation texture allows us to solve the problem of conforming the terrain shape to the vector features (see Figure 1): roads remain flat and river beds are properly defined. Our data structure elegantly captures data of very different natures – satellite imagery, elevation data and vector data from GIS – and enables efficient high-quality rendering of such data on modern programmable GPUs. In addition, because all the data is organized in the same quadtree structure, we can provide easy real-time editing of any graphical element in the terrain. We only have to perform local updates to the structure, hence ensuring interactivity.

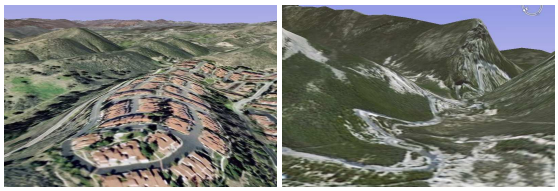


Figure 1: ***Two Google Earth views****. Despite the very high resolution, roads are not flat (*left*), rivers can appear to flow uphill (*right*), road markings and borders are blurred.*

Our contributions are the following:

- a data structure to store dynamic vector data at several LODs, that supports dense datasets (*e.g.*, fields);
- a GPU friendly scheme to generate not only texture, but also the shape of the terrain, using vector data;
- a method to incrementally update the vector data, including procedural data, allowing real-time terrain editing.

2. Related work

Our method relies on adaptive terrain rendering, fusion of geometrical data, and advanced shaders on GPU.

Terrain rendering Terrains can be provided using real data, precomputed data, or generated procedurally on the fly (this can also be used to amplify fixed data). Reviewing the extensive research work on terrain generation and real-time terrain rendering (ROAM [DWS*97], Geometry Clipmaps [LH04], C-BDAM [GMC*06], etc.) is beyond the scope of this paper. We use a classical view dependent quadtree refinement scheme, with a continuous LOD mechanism similar to that of the GPU adaptation of Geometry Clipmaps [AH05].

Combining terrain with secondary data Our goal is to populate terrains with features such as roads or rivers. In Geographic Information Systems (GIS), terrain data is often to be combined with other data (roads, buildings, etc.) separated for semantic, precision or memory reasons. Hence several methods have been proposed in the GIS community to combine the display of vector data with 3D terrains. Driving simulators also need similar techniques to draw roads. These methods can be divided into three approaches: overlay-geometry based, geometry based or texture based.

- The overlay-geometry based methods render 3D geometry on top of the terrain [WKW*03, ARJ06]. The difficulty is to keep these objects above the terrain, despite changes in the terrain mesh due to LOD. [SK07] avoids this problem with a type of shadow volume technique.
- The geometry-based methods insert the vector data into the terrain mesh itself. For instance, [WB01] and [PGJ95] integrate roads into a terrain mesh. But they perform this integration offline and do not deal with level of detail.
- The texture-based methods render the features as textures that are then mapped onto the terrain [KD02, Szo06]. This avoids the previous problems, but it requires a texture LOD management so that features remain sharp despite zooms. Our method is also inspired by this approach.

None of these methods but [PGJ95] deal with the modification of the terrain shape under the constraints introduced by the features. In addition, the interactive edition of these features requires dynamic terrain update. Most terrain LOD algorithms rely on some values precomputed from the terrain in a bottom-up way (e.g., for error bounds), so local modifications require complex updates. [HCP02] proposes an extension to the ROAM algorithm to account for this. This problem does not exist with the LOD mechanism of Geometry Clipmaps, which is thus well suited to our needs.

Adding features through shaders Textures can be used to account for details without adding more geometry, but for very large landscapes available texture memory can quickly be exhausted. Texture LOD management [TMJ98, LDN04] helps fitting the GPU memory, but shifts the cost to CPU memory and bus transfers. Procedural or semi-procedural textures are a solution to this, since they can be generated on the fly. [LN03] distribute and combine patterns to populate a large terrain with a detailed virtual texture. However their method is adapted to scattered features and to fields, not to long linear features such as road and rivers. [TC04, RBW04, RNCL05, QMK06, TC05] introduce methods to represent vector data such as printed characters in textures. However they require a precomputation phase to encode vector data into textures (a few seconds for pinchmaps [TC05]). In contrast, we simply adaptively refine spline curves on the CPU and rasterize them into textures using triangle strips. This allows us to generate vector-based textures in real time, with arbitrary shaders and attributes (u, v coordinates for curves, elevation profiles, etc.).

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

3. Overview

This section presents the overall organization of our system. The input data consists in a terrain DEM and a vector description of the terrain features. Everything else is generated on the fly, *i.e.*, the terrain shape and appearance, and the objects on top of it, all constrained by the vector features.

The central data structure is a quadtree on the CPU, dynamically refined based on the viewer's position (see Figure 15). Each quad contains the vector data appropriately clipped and refined for this quad, and an *elevation texture* that stores its shape. Each leaf in the current quadtree hierarchy, called a *leaf quad*, also contains an *appearance texture* that stores the terrain color and material, and some *object meshes* that represent the objects on top of it (trees, bridges, etc.). The vector data is stored on CPU, while textures and meshes are cached on GPU (see Figure 2). The terrain is rendered by drawing a flat regular grid for each leaf quad, with a vertex shader that uses the elevation texture to displace the vertices, and a fragment shader that uses the appearance texture. The object meshes are then drawn on top of it.

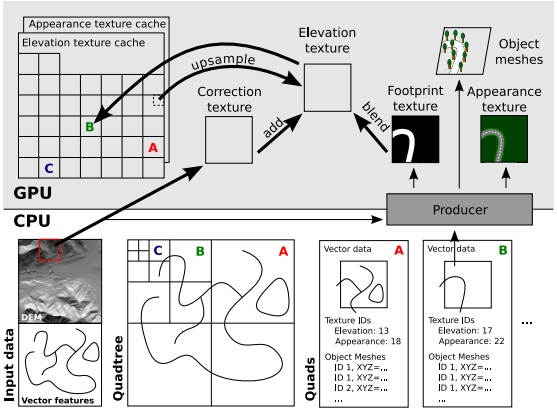


Figure 2: ***Overall organization***. *Production of the appearance and elevation textures of a quad (here B), and of its objects (here trees along the road) from its vector data. Note the repartition of data structures between CPU and GPU.*

When the viewer moves, some new quads must be created by subdividing existing quads, while others can be deleted. We create a new quad as follows:

- we compute the vector data by clipping and refining the vector data of the *parent* quad, with optional procedural elements (see Section 4);
- we generate the appearance texture on the GPU, by drawing the vector data into this texture, with the appearance shaders (see Section 5.1);
- we generate a footprint texture describing terrain height modifications similarly, and we blend it with the terrain height to get the elevation texture (see Section 5.2);
- we generate or instantiate the object meshes from the vector data with the object shaders (see Section 5.3).

When the user edits the vector data we update the quadtree from top to bottom. We update the vector data of each quad by recomputing only the necessary parts (see Section 6). If some changes have been made to a quad then we recompute its appearance and elevation textures and its object meshes. We support dense datasets by using a precomputed texture mipmap pyramid (see Section 7).

4. Data structures

4.1. Vector data

We aim at representing linear features such as roads, rivers or railways, and areal features such as lakes and fields of various types, with very precise shapes and decorations. To this end we use vector data, as in GIS applications, which allows for high precision, easy edition and use of existing data. More precisely we use *spline networks*: splines give smooth features at any resolution, and the network gives their *adjacency* relations (also called *topological* relations), which are necessary to draw specific road markings at crossings, for instance (see Figure 3).

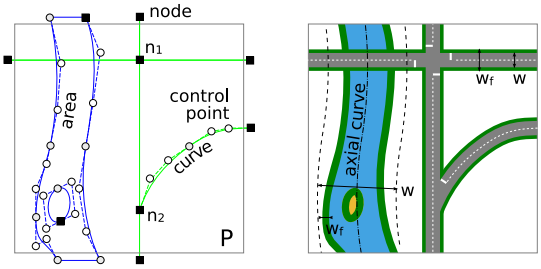


Figure 3: ***Data structures***. Left: *two* layers *(in blue and green) with their* graph *made of* nodes, curves *and* areas. Right: *the resulting* appearance texture. *Note that nodes with 3 or 4 adjacent curves (n_2 and n_1) result in different road markings (1 or 4 stop lines).*

Concretely, our vector data is made of several *layers* (*e.g.*, one layer for roads, another for rivers, etc.). Each layer is a *graph* that interconnects nodes, curves, and areas, called *graph elements* (see Figure 3):

- *nodes* are used for point features such as crossings and ends. They are specified by a 2D position and have references to their adjacent curves;
- *curves* are used for linear features such as roads and thin rivers. They are specified by a 2D Bezier spline curve that connects two nodes through a series of *control points*, a width w and a footprint width w_f (that includes roadsides, river banks, etc.). A curve has references to its end nodes and to the areas to which it belongs;
- *areas* are used for areal features such as large rivers, fields or forests. They are specified by a list of references to curves forming a loop (with possible holes), a footprint width w_f outside the surface, and an optional reference to

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

an axial curve of width w bounding the surface (used to define an elevation profile for the area – see Section 5.2).

4.2. Hierarchical structure

The CPU data structure is a dynamically refined quadtree. Each quad contains the vector data corresponding to its terrain part, computed on the fly by clipping and refinement. Hence the root quad contains all the vector data (except for large terrains – see Section 7). In the other quads the number of elements decreases with depth, each element being more and more refined. This structure brings several advantages:

- Using a quadtree is simpler than using a bintree or Geometry Clipmaps [LH04]. Indeed the L-shaped incremental updates of Geometry Clipmaps become a drawback when the terrain is generated from vector data. And bintrees slightly complicate clipping and texture management compared to quadtrees.
- Storing clipped data in each quad consumes memory but allows us to efficiently generate the terrain. Indeed, the textures of a quad q are generated by clipping the vector elements of its parent quad p (conservatively, *i.e.*, we just retain the Bezier arcs whose bounding box intersects q, taking the width w_f into account – see Figure 4), and by drawing the result. Assuming the number of vector elements is proportional to the quad area A, clipping and rendering are done in $O(A_p)$ and $O(A_q)$, *i.e.*, $O(A_q)$ since $A_p = 4A_q$. This still poses scalability issues for large quads (they are discussed in Section 7), but is much better than using all vector elements to generate any quad.
- Refining curves improves the precision of conservative clipping. And it allows us to draw the curves by using their polyline approximation. Indeed we use the de Casteljau algorithm to refine each curve, after clipping, until its distance to the polyline joining its control points is less than one pixel in the appearance texture (see Figure 4).

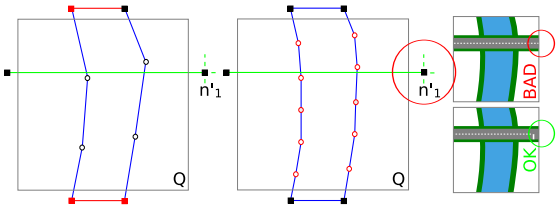


Figure 4: ***Clipping and refinement.*** Left: *clipping of Q, the topleft subquad of P (Fig. 3). Curves are clipped conservatively. New curves are added, in red, to close clipped areas.* Middle: *refinement.* Right: *the stop line before the (clipped) crossing is missed with n'_1's topology (1 adjacent curve). Using instead n_1's topology (4 adjacent curves) fixes this.*

Ancestor elements Each clipped graph has its own adjacency relations, which may differ from those of its parent graph. Since appearance depends on them (see Figure 3), using the adjacency relations of a clipped graph may produce errors (see Figure 4). In order to have access to the correct relations we store in each element a reference to its *ancestor* element, defined as the topmost element from which it derives through clipping and refinement only (either an element of the root quad, or a procedural element – see below).

4.3. Procedural graph elements

Some terrain features are constrained by others, *e.g.*, bridges must be placed where roads cross rivers. Others cannot be represented with a single curve or area, *e.g.*, a roundabout requires several arc circle curves between the connected road curves. These constrained and composed features are generally tedious to edit by hand. On the other hand they can often be generated automatically from a higher level description. We therefore provide a way to specify them procedurally.

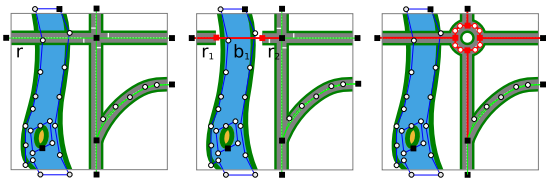


Figure 5: ***Procedural graph elements.*** *A graph (*left*) can be procedurally modified (in red) to create automatically curves for bridges (*middle*) or for roundabouts (*right*), for example.*

This is done with an optional, arbitrary graph transformation step before the clipping and refinement step, during a quad subdivision. This transformation can replace simple crossings with roundabouts, generate bridge curves where roads cross rivers, etc. (see Figure 5). It can also be used for other purposes, such as adding procedural details. For instance it can replace a curve with an area, to add details to thin rivers when the viewer is close to them. It can also simply load additional vector data from disk (for instance to load thin curves only when they become visible).

In summary the creation of a new quad can be described as follows (here for simplicity we ignore nodes, areas and layers, and consider only procedural bridges):

Algorithm 4.1: SUBDIVIDE(p, q)

procedure SUBDIVIDE(c, q)
  $\{c'\} \leftarrow$ BRIDGES(c)
  return $(\cup_{c \in \{c'\}}$ CLIPANDREFINE$(c,$ BOUNDS$(q)))$

main
  for each $c \in$ CURVES(p)
    do { CREATEDFROM$(q)[c] \leftarrow$ SUBDIVIDE(c, q)
           ADD$(q,$ CREATEDFROM$(q)[c])$
  GENERATEELEVATION(q)
  GENERATEAPPEARANCE(q)
  GENERATEOBJECTS(q)

where BRIDGES returns for each road curve r one or more

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

road and bridge curves $r'_1, b'_1, r'_2, b'_2, \ldots, r'_n$ (see Figure 5). It does so only if p is the root quad (like many procedural transformations, it does not make sense to apply it recursively at each subdivision). CREATEDFROM is used for incremental editing (see Section 6).

5. Texture and geometry producers

This section explains how we generate appearance and elevation textures, as well as object meshes, after the vector data has been clipped, refined and procedurally modified.

5.1. Appearance textures

The appearance texture of a quad defines the color and material of this terrain part. The basic idea for generating an appearance texture on the GPU is to draw the vector features into this texture using their associated appearance shaders (we can also load a satellite photograph or mix both – see Figure 14). For this, we draw 2D meshes based on the nodes, curves and areas of the quad. Curves are drawn using simple triangle strips of width w_f, constructed from the polyline joining the curve's control points. Areas are drawn using a triangulated mesh generated from the contour and hole curves, and a triangle strip of width w_f constructed outside along the surface border. Finally, nodes are drawn using triangle strips covering the end parts of the adjacent curves (see Figure 6).

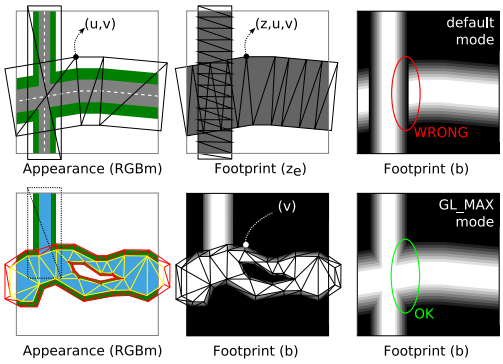


Figure 6: ***Appearance and footprint texture generation.*** Left: *curves are drawn using a single strip. Areas are drawn using an interior mesh (yellow) and a border strip (red).* Middle: *Footprint generation.* Right: *b coefficients are combined in* `GL_MAX` *mode to ensure continuity.*

Each element is drawn using its associated appearance shader, which depends on its type (road, river, lake, field, etc.). These fragment shaders can use the 2D position on the terrain, as well as the height and normal at this point (via a lookup in the elevation texture, which is generated first). For curves they can also use u, v coordinates (stored as vertex attributes) defined as the curvilinear coordinate along the curve, and the coordinate along its width. Finally they can use other parameters associated with the element (base color, road markings pattern, furrows direction for fields, etc.).

Curvilinear coordinates Consider a curve c clipped into two curves c_1 and c_2. Using local $u_1 \in [0..l_1]$ and $u_2 \in [0..l_2]$ coordinates on c_1 and c_2 would lead to a discontinuous u coordinate on c, yielding potential visible discontinuities between quads, *e.g.*, on road markings. It is therefore necessary to compute u on unclipped curves, *i.e.*, on *ancestor* curves (see Section 4.2). For efficiency we compute u only when necessary (*e.g.*, when the viewer is close enough to see the markings), and we cache the results for later use.

Uniform quads The appearance texture of a *uniform* quad, *i.e.*, a quad that is totally inside a single thick curve or a single area, is trivial to produce (just draw an OpenGL quad with the appearance shader). Then, instead of storing the appearance texture, we can just use the appearance shader when rendering the quad on screen. Doing so typically saves 50% of memory for close views, but requires more rendering time (since the appearance shader is called at each frame instead of only once).

5.2. Elevation and footprint textures

In order to have smooth and flat roads, rivers or lakes, we need to enforce the terrain height beneath them, with a smooth transition between the enforced area and the terrain (corresponding to roadsides, river banks, etc.). To ensure this we define the terrain height as the interpolation between an original height z_o and an enforced height z_e, using a blending coefficient b which varies from 1 inside the enforced area to 0 outside. z_e and b are specified with a *z-profile* $z(u)$ and a *blending profile* $b(v)$ based on the (u,v) coordinates of curves (see Figure 7). The interpolation is performed by blending a *footprint texture* containing z_e and b into an *elevation texture* containing z_o, with $z_m = bz_e + (1-b)z_o$.

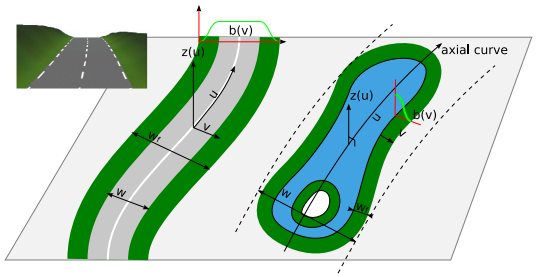


Figure 7: ***Z-profiles and blending profiles*** *are based on the (u,v) coordinates of the strip.* Top left inset: *$b(v)$ gives smooth roadsides.*

Z-profiles We want roads and rivers to have a smooth z-profile, while the terrain height around them may have higher frequencies. We also want to avoid rivers that flow uphill. Z-profiles must also adapt when the vector data is

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

edited. To this end we compute each z-profile from terrain samples $z_{o,i}$ regularly spaced along the curves. We then smooth them with a filter kernel w_j, *i.e.*, we compute $z(u)$ with $z(u_i) = z_{s,i} \equiv \sum z_{o,i+j} w_j$. We enforce decreasing altitudes for rivers by using $z(u_i) = min_{j \leq i}(z_{s,j})$. We also enforce z values at crossings so that adjacent curves have the same z at their junction. As for u coordinates, we cache these values and for efficiency we compute them only when necessary, on the ancestor curves (to avoid z-discontinuities).

Footprint texture This texture is generated by drawing the vector data into it. We draw curves in a single step using a fragment shader that produces both $z(u)$ and $b(v)$, based on $(u, v, z(u))$ vertex attributes (see Figure 6). We draw areas in three steps: we set $b \leftarrow 1$ inside the area by drawing the interior mesh, $b \leftarrow b(v)$ on the border by drawing the border strip, and $z_e \leftarrow z(u)$ in both regions by drawing a strip based on the axial curve of width w (with a stencil mask set in the previous steps). Terrain continuity requires continuous z_e and b channels. We ensure this while computing z-profiles for z_e, and by combining the continuous b values of each curve and area with a *max* function (see Figure 6).

Elevation texture In order to generate continuous transitions between quads of different levels, we store the terrain height at two successive resolutions, and interpolate linearly between the two during rendering, as in [LH04, AH05]. The elevation texture therefore contains 3 heights per texel: the original and modified height z_o and z_m, and the modified height at half resolution z_c. z_o and z_c are computed on the GPU as in [AH05], by upsampling the z_o component of the parent elevation texture, and adding a correction to it.

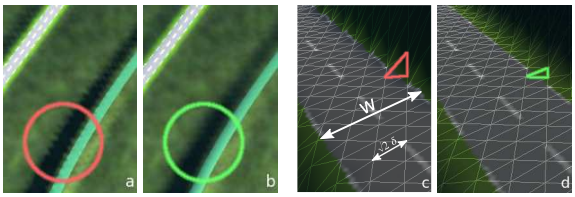


Figure 8: ***Elevation texture antialiasing***. Left to right: *geometric aliasing (a) is removed by filtering z_m values (b). Using the width w to rasterize footprint curves yields bad triangles at the border (c). Using $w + 2\sqrt{2}\delta$ instead solves the problem (d), with δ the size of an elevation texture pixel.*

Antialiasing The combined effect of the terrain slope, road width, z-profile and blending profile might generate spatial frequencies which cannot be represented at the current resolution, thus causing geometrical aliasing. To avoid this, we filter z_m values with a 3×3 pixel Gaussian filter in the elevation texture *where the blending coefficient b is not null*. This eliminates almost all aliasing effects without loosing details on the raw terrain (see Figure 8). We discard the footprint of curves such that $w < 2\delta$, where δ is the size of an elevation texture pixel (in meters), since they would not show up due to the previous filtering. We add twice the diagonal of a pixel $\sqrt{2}\delta$ to the width w to draw the remaining curves in order to force triangles crossing the curve's border to obey the z-profile (see Figure 8).

5.3. Object meshes

Object meshes are used for objects that are not part of the ground, such as bridges, hedges, buildings, etc. We generate or instantiate them from the vector data when a quad is needed. For instance we generate bridge meshes along bridge curves (one mesh per bridge), and we instantiate copies of a reference tree mesh along road curves at regular u intervals. For the same continuity reasons as in sections 5.1 and 5.2 this generation must be based on the ancestor curves, not on clipped ones (otherwise a bridge curve clipped in two parts would yield two bridges instead of one, for example).

6. Interactive editing

In many applications such as environmental impact studies or outdoor scene design for games or other applications, the user may want to edit the vector features. Using a 2D editor and a separate 3D viewer would be unintuitive and inefficient due to the lack of immediate feedback. We therefore want to provide real-time vector editing tools working directly on the 3D terrain view (see Figure 13). The difficult point here is to update our data structures in real time. For scalability reasons we need an algorithm whose cost is proportional to the number of *updated* elements, not to the total number N in the whole dataset.

Moving a control point can make a curve intersect a quad that was previously not intersected, leading to new clipped curves (and conversely – see R in Figure 9). It is therefore convenient to model all updates as sets of removed and added elements. We can then update the quadtree in a top-down way, from the root, with the following recursive procedure (we ignore nodes, areas and layers for simplicity):

Algorithm 6.1: UPDATE$(p, \{c_r\}, \{c_a\})$

if not ISLEAF(p) **and** $(\{c_r\} \neq \emptyset$ **or** $\{c_a\} \neq \emptyset)$
then for $i \leftarrow 0$ **to** 3
do
- $q \leftarrow$ SUBQUAD(p, i)
- $\{c'_r\}, \{c'_a\} \leftarrow \emptyset, \emptyset$
- **for each** $c \in \{c_r\}$
  - **do** $\{c'_r\} \leftarrow \{c'_r\} \cup$ CREATEDFROM$(q)[c]$; CREATEDFROM$(q)[c] \leftarrow \emptyset$
- REMOVE$(q, \{c'_r\})$
- **for each** $c \in \{c_a\}$
  - **do** CREATEDFROM$(q)[c] \leftarrow$ SUBDIVIDE(c, q); $\{c'_a\} \leftarrow \{c'_a\} \cup$ CREATEDFROM$(q)[c]$
- ADD$(q, \{c'_a\})$
- UPDATE$(q, \{c'_r\}, \{c'_a\})$

where $\{c_r\}$ and $\{c_a\}$ are curves that have been removed and

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

added in quad p. The outer loop body deletes the curves $\{c'_r\}$ in the subquad q that were created by clipping the curves $\{c_r\}$, and creates new clipped curves $\{c'_a\}$ for each c_a. It does so by using (and updating) the CREATEDFROM $c \rightarrow \{c'\}$ hash table. The resulting complexity is proportional to the number of updated elements, as desired. Note that procedural elements are automatically updated as well (see SUBDIVIDE(c,q) in Algorithm 4.1).

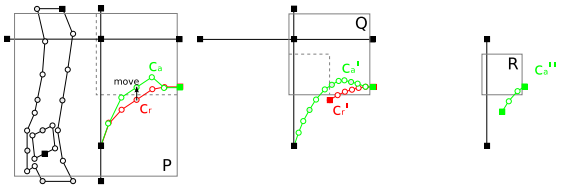


Figure 9: ***Incremental updates****. A road is edited in P. Q is P's topright subquad, R is Q's bottomleft subquad. Unchanged, removed and added curves are in black, red and green, respectively.*

Once the vector data has been updated, we recompute the curvilinear coordinates and the z-profiles of the ancestor curves that have changed, and we recompute from scratch the appearance and elevation textures and the object meshes of quads whose graphs have changed (this last step is in $O(N)$ in the worst case when the quadtree is reduced to a single quad, which rarely happens).

7. Scalability

The time to generate an appearance or footprint texture is proportional to the vector data size in the quad, which is itself approximately proportional to the quad area. This time can therefore become larger than the acceptable delay between two frames. The basic idea to solve this scalability problem is to use precomputed appearance and elevation textures for the first L levels of the quadtree (from the root). We then use vector data only for the other quads, and we choose L based on the terrain size and the vector density (assuming it is quite uniform). But this solution introduces two new tasks: computing and updating the precomputed textures, and managing and editing a vector data quadtree without its first L levels.

We precompute the textures for the first L levels as follows. We compute for each quad at level L, *and only them*, the corresponding clipped graphs, that we store on disk. We then generate the textures at level $L-1$ from these preclipped graphs. Finally we compute the textures for levels $L-2$ to 0 as for a mipmap pyramid. We store all these textures on disk. After this precomputation phase, when the vector data of a quad at level L is updated we recompute the textures of its ancestor quads in the same way and we update them on disk, as well as the updated clipped graphs.

The absence of vector data at the first L levels would yield the absence of ancestors for curves and areas that cross several quads at level L. However these ancestors are needed to ensure continuity (see Sections 4.2, 5.1, 5.2 and 5.3). Hence we use vector data at levels $0, L, L+1, \ldots$ (see Figure 10). If the vector data at level 0 does not fit in CPU memory it can be loaded lazily, *i.e.*, its elements can be loaded on first use, and unloaded after use. The required memory is then proportional to the maximum number of visible quads, which is logarithmic in the terrain size.

The vector data is updated as before, starting from the root level. We only need to replace the loop body in Algorithm 6.1 with UPDATE(SUBQUAD$(p,i),\{c_r\},\{c_a\}$) for levels less than L. The textures are then recomputed from the vector data for levels $\geq L$, and mipmapped in a bottomup way for levels $L-1$ to 0 (see Figure 10).

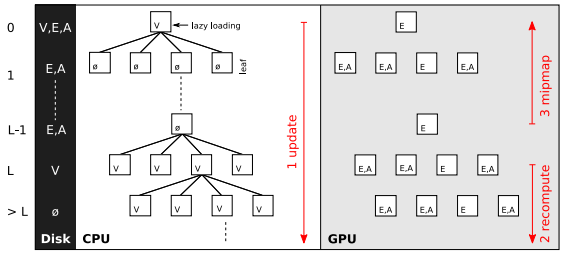


Figure 10: ***Scalability****. Storage and update of vector (V), elevation (E) and appearance (A) data.*

A similar scalability problem also exists for object meshes (trees, bridges, etc.): the number of objects can become too large to generate or render them in real time. Many strategies have been proposed to solve this problem for mass 3D data such as forests or cities. We consider it as out of the scope of this paper, which is focused on textural data.

8. Implementation

This section presents the implementation choices we used, in particular to get the results in the next section.

- We subdivide a quad if its distance to the viewer is less than K times its size. Thus a texture pixel covers at most $p = w_s/(2Kw_a \tan(\frac{1}{2}fov))$ pixels on screen, where w_s and w_a are the screen and appearance texture widths. We use $w_s = 1024$, $w_a = 192$, $fov = 80°$ and $K = 2.8$, which is sufficient to hide texture popping ($p = 1.1 \approx 1$).
- With $K = 2.8$ new quads tend to appear simultaneously, *e.g.*, 50 new quads can be requested for one frame, while no quads were generated during the 100 previous frames. This yields a strong temporal jittering. We greatly reduce this effect with a *prefetching* technique based on the predicted trajectory of the viewer, *i.e.*, we balance the computations needed for a frame on the 25 previous frames.
- The quadtree stores on CPU the vector data for all quads, either visible or not (we do not use lazy loading, since

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

our dataset fits in memory). On GPU, we store appearance textures only for leaf quads that are in the view frustum and not *uniform* (see Section 5.1). We store elevation textures only for internal and leaf quads that are in the frustum. This frustum culling works well for applications such as flight simulators. If the view can rotate quickly it is of course possible to store textures also for quads outside the view frustum.

- We use procedural bridge and roundabout curves. Currently these graph transformations are implemented manually in C++. Using a rule-based system to specify them would probably be possible but we did not investigate this.
- Real terrains support several materials whose BRDFs may differ a lot. The few systems that handle this use separate meshes or multipass rendering using masks. The first method is complicated, especially for real-time editing. We therefore use the second method, which also provides a correct filtering via the mipmapping of masks. We currently use only two materials: Lambert for the ground and Fresnel for the water. This allows us to keep one pass, one mesh and a single *RGBm* texture (3 channels for the color, one for the "water material" mask). Using more materials is possible with more texture channels.
- Appearance and elevation textures are stored either as tiles in a big texture on GPU, or in a texture array (on a GeForce 8800). We implemented both. Texture antialiasing and anisotropic filtering work better with texture arrays (no risk to mix samples that are adjacent in the texture cache but not on the terrain).
- Each appearance (resp. elevation) texture has a 192×192 (resp. 25×25) resolution. Our GPU caches can contain 500 appearance textures and 784 elevation textures. We restrict the prefetching window and/or degrade the quality of distant quads if needed (*i.e.*, we use the texture of a quad instead of the textures of its 4 subquads), so as to never exceed the cache capacity.

9. Results

Our terrain data covers a 100×100 km^2 area in the Alps, with a 5000×5000 DEM, 940k nodes and control points, 600k curves and 190k areas (including 400k nodes, 580k curves and 185k areas only for fields – see Figure 11). We tested two scenarios: a flyover and a terrain editing session. Both tests were conducted on an Intel Core 2 Duo 2.13 GHz (3 GB) with a GeForce 8800 GTS. Images are rendered at 1024×768 resolution with 8X anisotropic filtering.

9.1. Flyover

To measure the performance at various altitudes and speeds we use a flight path covering 13 km in 73 s, with a speed varying from 2000 km/h at 5000 m above the ground to 60 km/h at 8 m and 90 km/h at 5 m. In order to get a smooth animation we force the framerate to 60 fps, waiting between frames if necessary. This gives a 16.6 ms delay to compute each frame. We use it to generate the quads for this frame, and to prefetch some quads for the next 25 frames. If we have some time left we wait until the 16.6 ms are past (this time could be used for other computations). If the computations take more than 16.6 ms we adjust the position for the next frame to maintain the desired apparent speed (this works because these frames are rare – 2% – and not contiguous).

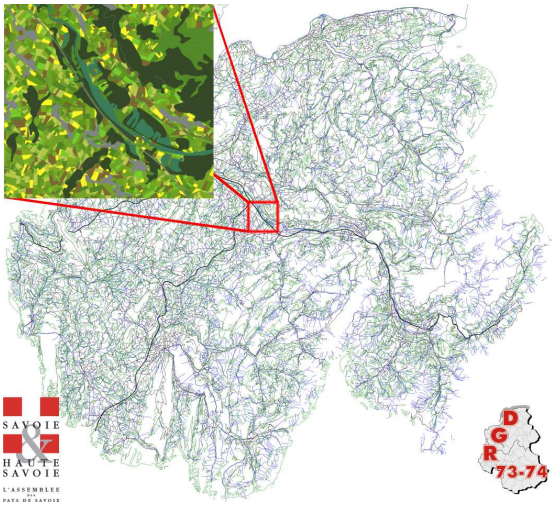


Figure 11: ***Test data****. A 100×100 km^2 area in the Alps, with a 5000×5000 DEM. The vector data is made of about 940k nodes and control points, 600k curves and 190k areas (60% of them are fields, not shown in wireframe view for clarity). Source:* Régie de Gestion des Données des Pays de Savoie.

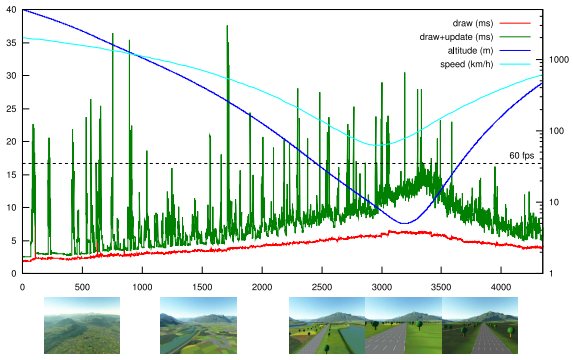


Figure 12: ***Performance****. Rendering time per frame during a flyover (see accompanying video). The total time (in green) includes the drawing time (in red) and the update time (difference between the two). The update time increases with the number of generated quads per frame, which increases with speed and proximity to the ground.*

In this experiment (and with a mipmap pyramid for the first $L = 7$ levels) the mean computation time per frame is 7.7 ms (hence leaving 8.9 ms for other computations), including 4 ms to draw the terrain, and 3.7 ms to update the data struc-

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

tures (vector data clipping, footprint and appearance tile generation – see Figure 12). The update time per frame varies from 0.6 to 25 ms for 99.8% of the frames (only 7 frames take more than 25 ms). This is because the number of generated footprint (resp. appearance) textures per frame varies from 0 to 9 (resp. 15) with an average of 1.49 (resp. 1.34), and because we currently have no precise oracle to guess the generation time for a given quad. Note that without prefetching the worst update time reaches 230 ms, with a maximum of 57 generated textures for the worst frame. At most 708 and 482 elevation and appearance textures are needed for the current frame. With the prefetched textures for the next 25 frames, this gives 781 and 500 textures (limited by the cache capacity). The maximum quadtree depth is 17 levels. This corresponds to a virtual $1.2\ 10^7 \times 1.2\ 10^7 = 192^2.4^{16}$ pixels appearance texture and $1.6\ 10^6 \times 1.6\ 10^6 = 25^2.4^{16}$ pixels elevation texture!

9.2. Interactive editing

In this experiment we use $L = 6$, we do not use prefetching, and we do not force any framerate (see Figure 13). We get interactive to real-time frame rates, depending on the situation (see accompanying video). Editing curves with associated curvilinear coordinates, z-profiles or object meshes, such as roads and rivers, costs more than editing other curves, such as forest or city area curves, because more data needs to be recomputed at each frame. Editing while looking down is also more efficient than in horizontal views, because fewer quads are visible and therefore fewer quads need to be recomputed at each frame.

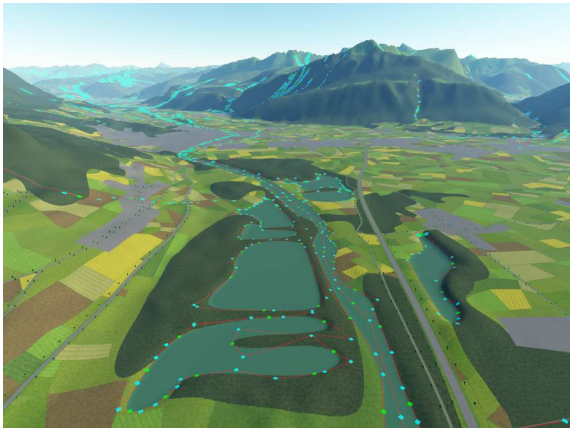


Figure 13: ***Interactive editing****. Each layer can be edited separately by moving the green and blue points, with immediate update. Here the rivers and lakes layer is edited.*

10. Conclusion

We presented a method able to handle vector features on very large landscapes, allowing for the real-time quality rendering of terrains with complex appearance (as demonstrated in the video): vector features appear at very high resolution, they can modify the terrain in order to enforce semantic constraints, and they can be edited interactively. We demonstrated various features: several kinds of roads, rivers, lakes, fields, crossings, bridges... Our representation can handle a mix of materials (ground and water), and support non-texture objects such as 3D bridges or trees. For future work, we would like to generalize the management of materials and the filtering quality. In terms of features, we would like to add more procedural controls for the aspect of surfaces, to handle non-flat vegetation, and to allow animation (e.g. for water surface).

Acknowledgments

This work was partially funded by the Natsim ANR ARA project. We would like to thank Cedric Manzoni, for the object meshes management and rendering implementation, as well as Antoine Bouthors, Sylvain Lefebvre and Jamie Wither for proofreading.

Figure 14: ***Results****.* Top: *large scale view.* Middle: *procedural bridges and roundabouts.* Bottom: *with our footprints (right) rivers are flat even if the raw elevation data lacks precision – which is often the case (left). NB: here the texture is a satellite photograph.*

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

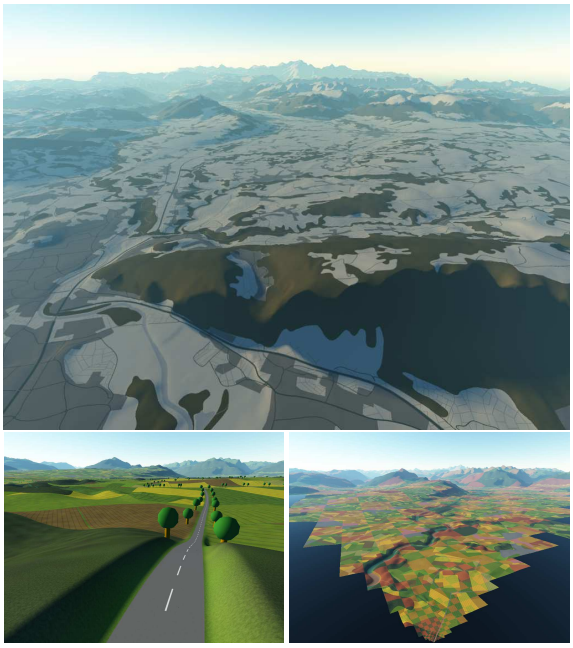


Figure 15: ***Other results***. Top: *a different rendering style.* Bottom: *a view (left) and the corresponding quadtree with frustum culling (right).*

References

[AH05] ASIRVATHAM A., HOPPE H.: *GPU Gems 2*. Addison Wesley, 2005, ch. Terrain rendering using GPU-based geometry clipmaps.

[ARJ06] AGRAWAL A., RADHAKRISHNA M., JOSHI R.: Geometry-based mapping and rendering of vector data over LOD phototextured 3D terrain models. In *Proceedings of WSCG* (2006).

[DWS*97] DUCHAINEAU M. A., WOLINSKY M., SIGETI D. E., MILLER M. C., ALDRICH C., MINEEV-WEINSTEIN M. B.: ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization* (1997), pp. 81–88.

[GMC*06] GOBBETTI E., MARTON F., CIGNONI P., DI BENEDETTO M., GANOVELLI F.: C-BDAM - compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum 25*, 3 (Sep 2006). Eurographics 2006 conference proceedings.

[HCP02] HE Y., CREMER J., PAPELIS Y.: Real-Time Extendible-Resolution Display of On-line Dynamic Terrain. In *Graphics Interface* (May 2002), pp. 151–160.

[KD02] KERSTING O., DÖLLNER J.: Interactive 3D visualization of vector data in GIS. In *ACM international symposium on advances in geographic information systems (GIS)* (2002), pp. 107–112.

[LDN04] LEFEBVRE S., DARBON J., NEYRET F.: *Unified Texture Management for Arbitrary Meshes*. Tech. Rep. RR5210, INRIA, May 2004. http://www-evasion.imag.fr/Publications/2004/LDN04.

[LH04] LOSASSO F., HOPPE H.: Geometry clipmaps: terrain rendering using nested regular grids. In *ACM SIGGRAPH* (2004), pp. 769–776.

[LN03] LEFEBVRE S., NEYRET F.: Pattern based procedural textures. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)* (2003).

[PGJ95] POLIS M. F., GIFFORD S. J., JR. D. M. M.: Automating the construction of large-scale virtual worlds. *Computer 28*, 7 (1995), 57–65.

[QMK06] QIN Z., MCCOOL M. D., KAPLAN C. S.: Real-time texture-mapped vector glyphs. In *ACM-SIGGRAPH Symposium on Interactive 3D graphics and games (I3D)* (2006), pp. 125–132.

[RBW04] RAMANARAYANAN G., BALA K., WALTER B.: Feature-based textures. *Eurographics Symposium on Rendering* (2004), 65–73.

[RNCL05] RAY N., NEIGER T., CAVIN X., LEVY B.: Vector texture maps. In *Tech Report* (2005). http://alice.loria.fr/publications/papers/2005/VTM/vtm.pdf.

[SK07] SCHNEIDER M., KLEIN R.: Efficient and accurate rendering of vector data on virtual landscapes. In *Proceedings of WSCG* (2007).

[Szo06] SZOFRAN A.: Global terrain technology for flight simulation. In *Game Developers Conference* (2006).

[TC04] TUMBLIN J., CHOUDHURY P.: Bixels: Picture samples with sharp embedded boundaries. In *Rendering Techniques* (2004), Keller A., Jensen H. W., (Eds.), Eurographics Association, pp. 255–264.

[TC05] TARINI M., CIGNONI P.: Pinchmaps: textures with customizable discontinuities. *Computer Graphics Forum 24*, 3 (2005), 557–568.

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

[WB01] WEBER A., BENNER J.: Interactive generation of digital terrain models using multiple data sources. In *First International Symposium on Digital Earth Moving (DEM)* (2001), Springer-Verlag, pp. 60–64.

[WKW*03] WARTELL Z., KANG E., WASILEWSKI T., RIBARSKY W., FAUST N.: Rendering vector data over global, multi-resolution 3D terrain. In *Eurographic symposium on Data visualisation (VISSYM)* (2003), pp. 213–222.

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.