Patchwork Terrains: Multi-Resolution Representation from Arbitrary Overlapping Grids with Dynamic Update

Luigi Rocca[1], Daniele Panozzo[2], and Enrico Puppo[1]

[1] DIBRIS, University of Genoa, Genoa 16146, Italy,
{rocca,puppo}@disi.unige.it
[2] IGL, ETH Zurich, Zurich, Switzerland,
panozzo@inf.ethz.ch

Abstract. We present a radically new method for the multi-resolution representation of large terrain databases. Terrain data come as a collection of regularly sampled, freely overlapping grids, with arbitrary spacing and orientation. A multi-resolution model is built and updated dynamically off-line from such grids, which can be queried on-line to obtain a suitable collection of patches to cover a given domain with a given, possibly view-dependent, level of detail. Patches are combined to obtain a C^k surface, with k depending on the type of base patches. The whole framework is designed to take advantage of the parallel computing power of modern GPUs.

1 Introduction

Management of huge terrain datasets is a challenging task, especially for virtual globes, like Google Earth and Microsoft Virtual Earth, and GIS modules performing analyses in hydrography, land use, road planning, etc. In fact, such applications may need to cope with terabytes of data.

Digital Elevation Maps (DEMs) consist of collections of grids, which may have different resolutions and different orientations. In order to support interactive data manipulation, it is necessary to rapidly fetch a suitable and properly organized subset of data, which is relevant for the problem at hand. Continuous Level Of Detail (CLOD) models support dynamic extraction of representations for a given domain at a given accuracy. However, to the best of our knowledge, all CLOD models in the literature are based on static data structures that cannot be updated dynamically [12].

In this paper, we present an approach to CLOD terrain modeling that is radically different from previous literature. Its salient features can be summarized as follows (see Figure 1):

1. Our method provides on-line a compact C^k representation of terrain at the desired accuracy over a given domain, with a degree of smoothness k selected depending on application requirements. This representation is obtained by

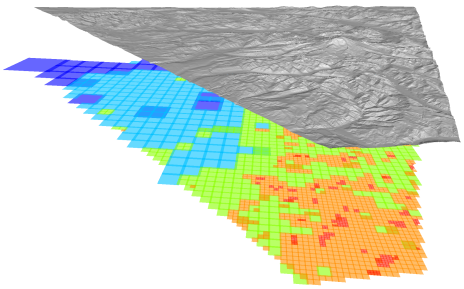


Fig. 1. A view-dependent query executed on the Puget Sound Dataset with an on-screen error of one pixel. The wedge is the portion of domain intersected by the view frustum for an observer placed above its apex. The different colors represent patches of different sizes.

blending a collection of freely overlapping rectangular patches, of different sizes and orientations, which locally approximate different zones of terrain at different detail.

2. Starting from the input DEMs, we produce a large collection of small patches of different sizes and accuracies, and we store them in a spatial data structure indexing a three-dimensional space. Such embedding space has two dimensions for the spatial domain, and a third dimension for the approximation error. Every patch is represented as an upright box: its basis corresponds to the domain covered by the patch; its height corresponds to the range of accuracies for which the patch is relevant. We optimize the range of accuracies spanned by each patch, so that the number of patches used to represent a given LOD is minimized. Independent insertion of patches in the spatial index can be performed easily and efficiently, and the result is order independent, thus dynamic maintenance of the database is supported.
3. CLOD spatial queries are defined by specifying a surface in the embedding space, which encodes space culling and detail requirements altogether. Such queries are executed on-line by finding the set of boxes that intersect this user-defined surface.
4. The extracted representation can be resampled on-the-fly to produce an adaptive (possibly view-dependent) tessellation with arbitrary connectivity.

This paper describes the general framework, alongside with two proof-of-concept implementations. The first implementation provides a C^0 representation that can be efficiently sampled in real-time in the GPU. The second implementation provides a smooth C^2 representation which and can be used for computationally intensive GIS tasks. We present results obtained on a moderately large dataset containing about 256M points.

2 Related work

Overall, known approaches to terrain modeling and rendering can be subdivided into three categories, reviewed in the following. The first category is better suited for modeling purposes, while the other two categories are specifically designed for rendering. Our proposal belongs to none of them, and it can be tailored to both rendering, and other GIS tasks.

CLOD refinement. These methods produce triangle meshes, which approximate terrain according to LOD parameters that can vary over the domain. They are mostly used for modeling and processing purposes, since they provide an explicit representation, with the desired trade-off between accuracy and complexity. Specific CLOD methods, tailored for rendering, build clusters of triangles in a preprocessing steps, possibly at different resolutions [2, 4, 6]. Clusters are selected on-the-fly at rendering time and passed to the GPU in batches: the rendering primitive is not anymore a single triangle, but rather a triangle strip encoding a large zone of terrain. Recent surveys on CLOD refinement methods can be found in [12, 20].

Geometry Clipmaps. In the approach presented in [9], a set of nested regular grids centered about the viewer are stored in the GPU memory, and used for rendering. As the viewpoint moves, the Geometry Clipmaps are updated in video memory. Tessellation is performed directly in the GPU. This method requires that input comes as a single uniform grid at high resolution, and it takes advantage of the intrinsic coherence of height maps to compress the input, thus reducing the amount of data that are passed to the GPU. Very high frame rates can be obtained, even for huge datasets.

GPU Ray-Casting. The use of ray-casting for rendering height maps is well studied in the literature, and different GPU techniques that achieve real-time frame rates have been developed in recent years. Methods for real-time rendering of meshes and height maps represented as Geometry Images have been proposed in [3, 11, 15]. However, all these methods were not designed to work with large terrain datasets. In [5], a tiling mechanism is used to support real-time rendering on arbitrarily large terrains. Ray-casting methods can be used only for the purpose of rendering, since they do not produce an explicit multi-resolution representation. In [16] a method is proposed, which exploits fast GPU wavelet mechanisms to support both ray-casting rendering and interactive editing of huge terrain datasets. Also this method requires input data to come as a single regular grid at high resolution. In [1], a hybrid approach that combines ray casting and mesh-based rendering has been proposed.

3 Patchwork terrains

In this Section, we describe our technique: in Subsection 3.1, we define the type of patches we use; then, in Subsection 3.2, we describe how patches are blended

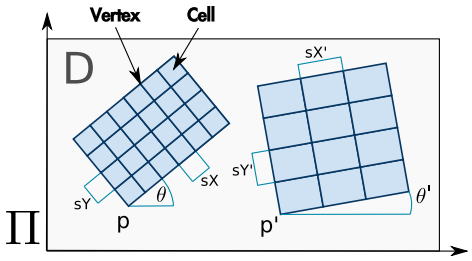


Fig. 2. The terrain is covered by a set of regularly sampled grids. Every grid has its own anchor point p, orientation angle θ and different sample steps for the two axes sX and sY.

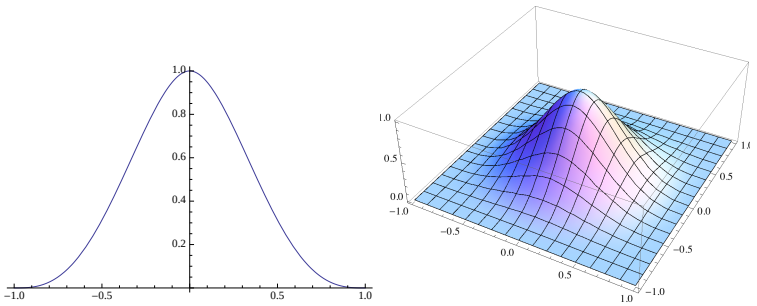
to form a C^k representation of terrain; finally, in Subsection 3.3, we describe the multi-resolution model, the order-independent algorithm for the dynamic insertion of patches and the spatial queries.

3.1 From grids to patches

We assume a two-dimensional global reference system Π on which we define the domain D of the terrain, where all input grids are placed. A grid is a collection of regularly sampled height values of terrain. In addition to the matrix of samples, every grid is defined by an anchor point, an angle that defines its orientation, and grid steps in both directions (see Figure 2). In the following, we will use the term *vertex* to denote a sample point on the grid, and the term *cell* to denote a rectangle in D spanned by a 2×2 grid of adjacent vertices. The *accuracy* of a grid also comes as a datum, and it is the maximum error made by using the grid to evaluate the height of any arbitrary point on terrain.

We aim at defining parametric functions that represent small subsets of vertices of the grid, called *patches*. A single patch is defined by an anchor point, its height, its width, and the coefficients that describe the parametric function. For the sake of simplicity, we will consider the height and width of every patch to be equal, hence the domain of every patch will be a square. Extension to rectangular patches is trivial.

We consider two types of patches: *perfect* patches interpolate the samples of the original terrain; while *approximating* patches represent the terrain ad a lower level of detail and accuracy. We assign an error to each patch, namely the accuracy ε of the input grid for a perfect patch; and $\varepsilon + \delta$ for an approximating patch, where δ denotes the maximum vertical distance between the input grid and the approximating patch. We will denote as *kernel* a rectangular region inside every patch, while the rest of the patch will be denoted as its *extension zone*. The extension zone will be used for the purpose of merging different patches, and the ratio between the sizes of the kernel and of the extension zone provides a trade-off between efficiency and smoothness of transition between different patches, which will be clarified later on.



(a) (b)

Fig. 3. C^2 weight functions: (a) 1D function w, plotted between -1 and 1. (b) 2D function W, plotted with x,y between -1 and 1, with the parameter d set to 1.

The type of function defining a patch will vary depending on the application. In Section 4.1 we provide specific examples. Our technique, however, can be used with any kind of parametric function: depending on the application, it may be convenient to use either a larger collection of simpler patches, or a smaller collection of more complex patches. The rest of this section is generic in this respect.

Note that, unlike splines, our patches may freely overlap, without any fixed regular structure.

3.2 Merging patches

Given a collection of freely overlapping patches, we blend them to produce a smooth function that represents the whole terrain spanned by this collection. In order to obtain a C^k surface that is efficient to evaluate, we use a tensor product construction, starting from the one dimensional, compactly supported radial basis function defined in [21]. Our weight function is defined as:

$$W(x, y, d) = \frac{w(x/d)w(y/d))}{\int_{-1}^{1}\int_{-1}^{1} w(x/d)w(y/d)\ dxdy}$$

for $x, y \in [-d, d]$ and 0 elsewhere. The 1-dimensional weight $w(t)$ is a C^k function with compact support, as defined in [21]: see Figure 3 for the C^2 case and Section 4.1 for further details. It is easy to see that the weight function W has the following properties:

1. It has compact support in $[-1, 1] \times [-1, 1]$;
2. Its derivatives up to order k vanish on the boundary of its support;
3. It is C^k in $[-1, 1] \times [-1, 1]$;
4. It has unit volume.

The first three conditions guarantee that the weight function has limited support, while being C^k everywhere. This is extremely important for efficiency

reasons, as we will see in the following. Property 4 is useful, since it naturally allows smaller (and more accurate) patches to give a stronger contribution to the blended surface.

For every patch P, we define its weight function $W^P(x, y)$ as a translated and scaled version of W, such that its support corresponds with the domain of P:

$$W^P(x, y) = W(|x - P_x|, |y - P_y|, P_s)$$

with P_x and P_y the coordinates of the center of P and P_s the size of P.

A collection of C^k patches $P^1, P^2, ..., P^n$ placed on a domain D, such that every point of D is contained in the kernel of at least one patch, defines a C^k surface that can be computed using the following formula:

$$f(x, y) = \frac{\sum_{i=0}^{n} P_f^i(x, y) W^{P^i}(x, y)}{\sum_{i=0}^{n} W^{P^i}(x, y)} \tag{1}$$

with P_f^i the function associated with patch P^i.

Note that the surface is C^k inside D since it is defined at every point as the product of C^k functions and the denominator can never vanish since every point in D belongs to the interior of the domain of at least one patch. The summation actually runs only over patches whose support contains point (x, y), since the weight function will be zero for all other patches.

At this point, terrain can be described with an unstructured collection of patches. To use this method on large datasets, we still miss a technique to efficiently compute this representation at a user-defined LOD.

3.3 The multi-resolution model

We build a multi-resolution model containing many patches at different LODs, each patch being defined by a small number of parameters, and we provide a simple and efficient algorithm to extract a minimal set of patches covering a given region of interest at a given LOD, possibly variable over the domain.

We define a 3D embedding space, called the *LOD space*, in which two axes coincide with those ones of the global reference system Π, while the third axis is related to approximation error. For simplicity, we will set a maximum allowed error, so that LOD space is bounded in the error dimension. In this space, every patch will be represented as an upright box (i.e., a parallelepiped), having its basis corresponding to the spatial domain of the patch, and its height corresponding to the range of approximation errors, for which the patch is relevant. The bottom of the box will be placed at the approximation error of the patch, while its top will be set to a larger error, depending on its interaction with overlapping boxes, as explained in the following.

In this section, patches will be always treated as boxes, disregarding their associated functions. We will consider open boxes, so that two boxes sharing a face are not intersecting. For a box B, we will denote as $B.min$ and $B.max$ its

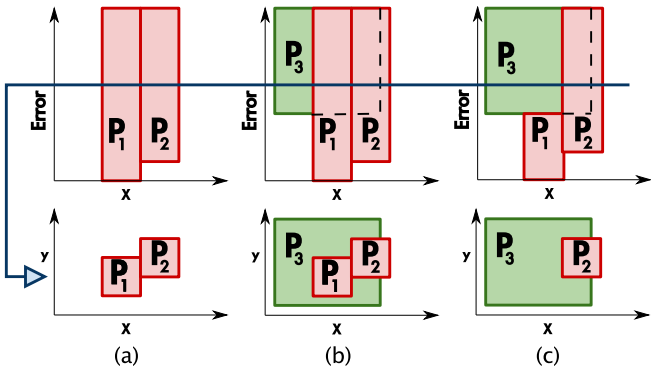


Fig. 4. Boxes of patches in LOD space, with a cut shown by the blue line: (a) Two independent patches P_1 and P_2; (b) A third patch P_3 is added: patch P_1 becomes redundant for the given cut; (c) P_1 is shortened to obtain a minimal set of patches for every cut of the spatial index.

corners with minimal and maximal coordinates, respectively. Furthermore for a point p in LOD space, we will denote its three coordinates as $p.x$, $p.y$ and $p.z$.

Given a collection of patches embedded in LOD space, a view of terrain at a constant error e can be extracted by gathering all boxes that intersect the horizontal plane $z = e$. More complex queries, which may concern a region of interest as well as variable LOD, can be obtained by cutting the LOD space with trimmed surfaces instead of planes.

To informally describe our approach, let us consider the examples depicted in Figure 4. Figure 4(a) shows two non-overlapping patches embedded in LOD space: P_1 is a perfect patch with zero error, and its box extends from zero to maximum error in the LOD space. This means that P_1 will be used to approximate its corresponding part of terrain at all LODs. On the contrary, patch P_2 is an approximating patch: it has its bottom set at its approximation error, while its top is again set at maximum error. Patch P_2 will be used to represent its part of terrain at any error larger or equal than its bottom, while it will not be used at finer LODs.

In Figure 4(b), a larger patch P_3 is added to our collection, which has a larger error than P_1 and P_2 and also it completely covers P_1. A cut at an error larger than the error of P_3 would extract all three patches, but P_1 is in fact redundant, since its portion of terrain is already represented with sufficient accuracy from P_3, which also covers a larger domain. In order to obtain a minimal set of patches, in Figure 4(c) patch P_1 is shortened in LOD space, so that its top touches the bottom of P_3. Note that we cannot shorten P_2 in a similar way, because a portion of its spatial domain is not covered by any other patch.

This simple example leads to a more complete invariant that patches in LOD space must satisfy to guarantee that minimal sets are extracted by cuts. We first formally describe this invariant, then we provide an algorithm that allows us to fill the LOD space incrementally, while satisfying it. This algorithm builds the

multi-resolution model and the result is independent of the order of insertion of patches. Implementation will be described later in Section 4.2.

We define a global order $<$ on patches as follows: $P < P'$ if the area of P is smaller than the area of P'; if the two areas are equal, then $P < P'$ if $P.min.z > P'.min.z$, i.e., P is less accurate than P'.

Since both the spatial extension and the approximation error of a patch P are fixed, the spatial invariant is only concerned with the top of P, i.e., with its maximal extension in the error dimension.

Patch Invariant: A patch P must not intersect any set of patches, such that the union of their kernels completely covers the kernel of P, and each patch is greater than P in the global order $<$. Also, the patch P cannot be extended further from above without violating the previous condition.

In other words, this invariant states that a patch is always necessary to represent terrain at any LOD, in its whole extension in the error domain, because that portion of terrain cannot be covered by larger patches. If all the patches in the model satisfy this property, we are sure that we will obtain a minimal set of patches whenever we cut the model with horizontal planes of the form $z = c$. The second part of the invariant enforces patches to span all levels of error where their contribution is useful for terrain representation, thus maximizing the expressive power of the model. More general cuts will also extract correct representations in terms of LOD, but minimality is not guaranteed.

Let us consider inserting a new patch P into a collection of patches that satisfy the invariant. If the new patch does not satisfy the invariant, we shorten it at its top. This is done through Algorithm 1 described below. Note that a patch may be completely wiped out by the shortening process: this just means that it was redundant. After the insertion of P, only patches that intersect P may have their invariance property invalidated, so we fetch each of them and we either shorten or remove it, again by Algorithm 1. All this process is done through Algorithm 2. Shortening patches that were already in the model does not invalidate invariance of other patches, so no recursion is necessary.

It is easy to see that all patches in a model built by inserting one patch at a time through Algorithm 2 satisfy the invariant. We also show that the result is independent on the order patches were added.

Order Independence: The structure of a model built by repeated application of Algorithm 2 is independent of the insertion order of patches.

Proof: The height of the box associated to a patch depends only on the spatial position and minimal error of the other patches inserted in the spatial index. The invariant guarantees that all boxes have their maximum allowed size in the error dimension, with respect to all other patches in the model. Therefore, the final result only depends on what patches belong to the model. □

To summarize, the algorithm shown allows us to dynamically build and update a spatial data structure that automatically detects and discards redundant data. Queries are executed on-line by cutting such structure with planes or surfaces. Extracted patches are merged, as explained in Section 3.2, to produce the final terrain representation.

Algorithm 1 cutter(Patch P, SetOfPatches ps)

```
1: sort ps in ascending order wrt min.z
2: current = {}
3: last = {}
4: for P' ∈ ps do
5:    if P ¡ P' then
6:       current = current ∪ {P'}
7:       if the patches in current cover P then
8:          last = P'
9:          break
10:      end if
11:   end if
12: end for
13: if not (last == {})  then
14:   if last.min.z ¡ P.min.z then
15:      Remove P
16:   else
17:      P.max.z = P'.min.z
18:   end if
19: end if
```

Algorithm 2 add-patch(Patch P)

```
1: ps = patches that intersect P
2: cutter(P, ps)
3: for P' ∈ ps do
4:    if P' still intersects P then
5:       ps' = patches that intersect P'
6:       cutter(P', ps')
7:    end if
8: end for
```

This completes the theoretical foundations of our technique. We discuss the implementation details in Sections 4 and 5, while we provide benchmarks and results in Section 6.

4 Implementation of the spatial index

This section describes a possible implementation of the general framework presented in Section 3, which has been kept as simple as possible for the sake of presentation. In Section 4.1 we describe the construction of patches, while in Section 4.2 we describe the implementation of the spatial index.

4.1 Generation of Patches

We describe two types of patches: bilinear patches provides a C^0 representation of the terrain that can be used for rendering purposes; while bicubic patches provide a C^2 representation, trading speed for increased terrain quality.

We use patches at different scales, which are generated from sub-grids of the various levels of a mipmap of terrain data. Each patch is a rectangle that covers a set of samples of the terrains. The patch must represent the terrain it covers, and its size depends on the density of grid samples. Patches may also cover mipmaps, thus allowing to represent larger zones of the terrain with less samples.

For every level of the mipmap, we build a grid of patches such that the union of their kernels form a grid on the domain, and the intersections of their kernels are empty. The size of the kernel with respect to the size of the patch is a parameter controlled by the user, that we denote σ. Any value $0 < \sigma < 1$ produces a C^k terrain representation; different values can be used to trade-off between quality and performance: small values of σ improve the quality of blending between patches; conversely, large values reduce the overlapping between different patches, thus improving efficiency, but transition between different patches may become more abrupt, thus producing artifacts. In our experiments, we obtained satisfactory results by using $\sigma = 0.9$.

Bilinear patches are formed by a grid of samples and they are simply produced by bilinear interpolation of values inside every 2x2 sub-grid of samples. These patches are C^0 in their domain, and the blending function we use is $w(t) = (1 - |t|)$.

Bicubic patches are formed by a grid of samples, as in the case of bilinear patches. To define a piecewise bicubic interpolating function we compute an interpolant bicubic spline with the algorithm described in [13]. These patches are C^2 in their domain, and the blending function we use is $w(t) = (1 - |t|)^3(3|t| + 1)$.

4.2 Spatial Index

The spatial index must support the efficient insertion and deletion of boxes, as well as spatial queries, as explained in Section 4.3. An octree would be an obvious choice, but it turns out to be inefficient, because large patches are duplicated in many leaves. We propose here a different data structure that is more efficient for our particular application.

Given a patch P, we define its *z-span* to be the interval $[P.min.z, P.max.z]$ of errors for which P is relevant, and the *z-ceiling* of P to be the highest value of its z-span. We build a quadtree over the first two dimensions. For the sake of brevity, we will refer with the same symbol q to a node in the quadtree and to its related quadrant in the spatial domain. We store at every node q (either internal or leaf) a set of patches that intersect the domain of q. Not all intersecting patches are stored, but just the first t patches that have the highest z-ceilings,

and that are not stored in any ancestor node of q. We use a threshold t of 64 in our experiments. There is no guarantee that a patch is stored in exactly one node, but in our experiments a patch is always stored in less than two nodes on average. We define the *z-span* of node q to be the smallest interval that contains th union of all patches stored at q.

The quadtree fulfills the following invariant: *for a quadrant q of the quadtree, let* $[z_q, Z_q]$ *be its span, then: all patches intersecting q and having a z-ceiling larger than* Z_q *are stored in the ancestors of q; all patches intersecting q and having a z-ceiling between* z_q *and* Z_q *are stored in q; and all patches intersecting q and having a z-ceiling smaller than* z_q *are stored in the children of q*. This kind of structure is similar to Multiple Storage Quadtrees [14] and it can be exploited to support spatial queries, as explained in the following subsection.

Inserting a new box in the tree is simple. Starting at the root, a box B is inserted in the node(s) that intersects its spatial domain, if and only if either the number of patches in such node does not exceed its capacity, or the z-ceiling of the new box is larger than the z-ceiling of the last box in the list at that node; in the latter case, the last box of the list (which has the minimum z-ceiling in the list) is moved downwards in the tree. Otherwise, the new patch is moved downwards in the tree.

4.3 Spatial queries

As explained in Section 3.3, queries are specified by a surface in LOD space. The projection of such a surface in the spatial domain is the region of interest (ROI) of the query, which will drive traversal of the quadtree. The z-values of the surface define the error tolerance at each point in the ROI, and will provide thresholds to prune the search.

At query time, the quadtree is traversed top-down, and quadrants that intersect the ROI are visited. For each such quadrant q, its z-span $[z_q, Z_q]$, is compared with the z-interval $[z_t, Z_t]$ spanned by the portion of query surface intersecting q. If $Z_q < z_t$, then the search is pruned at q; if $z_q > Z_t$ then the patches stored at q are discarded and the search is propagated to the children of q; otherwise the list of patches is scanned and a patch P is selected if and only if its z-span intersects the z-interval spanned by the query surface on the domain of P. Traversal of the list can be interrupted as soon as a patch having a span that does not intersect interval $[z_t, Z_t]$ is found (as that patch, and all subsequent patches, are more accurate than needed).

View-Dependent Queries. For applications such as view-dependent rendering, the accuracy of the extracted representation should smoothly decrease with distance from the viewpoint, so that larger patches can be used on far portions of terrain, thus reducing the computational load, without introducing visual artifacts. We perform a view-dependent query by cutting the LOD space with a skewed plane, which aims at extracting a model with a constant screen error, for a given viewpoint.

In [7] a method was proposed that computes the maximum error in world coordinates that we can tolerate, in order to obtain an error in screen coordinates smaller than one pixel. Such a method defines a surface in LOD space that we could use to make view-dependent queries in our spatial index. However, the resulting surface is complex and the related intersection tests would be expensive. We use an approximation of such a method that allows us to cut the spatial index with a plane, which provides a conservative estimate of the correct cutting surface: we obtain a surface that is correct in terms of screen error, while it could be sub-optimal in terms of conciseness.To compute the cutting plane, we ignore the elevation of the viewer with respect to the position of the point, obtaining the following formula:

$$\delta_{screen} = \frac{d\lambda\delta}{\sqrt{(e_x - v_x)^2 + (e_y - v_y)^2}},$$

with e being the viewpoint, v the point of the terrain where we want to compute the error, d the distance from e to the projection plane, λ the number of pixels per world coordinate units in the screen xy coordinate system, δ the error on world coordinate and δ_{screen} the error in pixels.

This plane is reduced to a triangle by clipping the zones outside the view frustum. The spatial index is then cut with this triangle, and the intersection between boxes in the index and the triangle are efficiently computed with the algorithm of [19], after an appropriate change of reference system has been performed on the box.

5 Terrain tessellation

In this section, we discuss in detail how a terrain represented as a collection of patches, as extracted from the spatial index, can be resampled to obtain a tessellated model. We describe general principles concerning the resampling operation, and we provide a CPU and a GPU implementation.

So far, we have shown how to extract a parametric C^k representation of terrain at the desired LOD from the spatial index. Let G be a grid on the domain of terrain and let S be the set of extracted patches. To render the terrain, we rasterize it by imposing G on the domain and by evaluating the parametric surface only at its vertices, using equation (1). The computation can be performed efficiently by observing that the weight function associated with a patch P is zero for all the vertices of G that lies outside the domain of P. Thus, for every patch P^i in S, we need to evaluate P^i_f and W^{P^i} just for the vertices of G that lie in the domain of P^i.

Note that sampling the terrain at a certain coordinate is an indipendent operation. We use grids only for convenience - irregular triangulations could be used just as easily. In the following paragraphs, we suggest two simple but effective ways to support uniform as well as view-dependent resampling.

Uniform resampling: A uniform rendering is easily obtained by imposing a regular grid on the terrain domain. Note that this resampling operation is

decoupled from the desired LOD, already obtained by querying the spatial index, and can be tailored to application needs.

View-dependent resampling: A view-dependent rendering is obtained by imposing a position-dependent grid on the terrain domain. We produce a grid in screen space that has approximately the same number of samples as the number of pixels on the screen. By projecting this grid on the terrain domain we obtain a trapezoidal grid with a high density of vertices in the neighborhood of the viewer, and progressively lower densities as we move farther. This technique is similar to the Persistent Grid Mapping proposed in [8].

5.1 CPU resampling

To efficiently perform the resampling operation in CPU, we have built a two-dimensional spatial index on the domain on the terrain. This spatial index contains the position of all vertices of G and allows us to rapidly fetch all vertices contained in the domain of a patch, exploiting the fact that only a small subset of patches in S has a non-zero contribution to a particular vertex. Note that this spatial index has to be built just once, since the grid is uniform or depends only on the position of the viewer. If a viewer moves in subsequent frames, we do not move the grid, but we rather translate and rotate the patches returned by the query to place the grid in the desired position. By using this spatial index, we can efficiently extract the vertices that lie in every patch and incrementally compute Equation (1). Note that the implementation can be easily parallelized on multi-core CPUs with a multi-thread implementation, since every vertex can be sampled independently.

5.2 GPU resampling

We have developed an experimental GPU implementation using the nVidia CUDA language. It works using vertices as parallelization points. Every GPU thread resamples a vertex and runs through all extracted patches, searching for the relevant ones. Despite still being a basic prototype, results are promising (as shown in section 6). Ideas for a complex GPU data structure specifically tailored to the patchwork model are discussed in section 7.

6 Results

In this Section we present the results obtained with our prototype implementation on a dataset over the Puget Sound area in Washington. Experiments were run on a PC with a 2.67Ghz Core i5 processor equipped with 8Gb of memory and a nVidia GTX275 graphic card. The dataset is made up of 16,385 $\times$ 16,385 vertices at 10 meter horizontal and 0.1 meter vertical resolution [17]. Our framework produces a single frame to be rendered in two main phases: a query to the spaial index, in order to obtain a set of patches representing the terrain at the desired LOD; and a tessellation phase, where points on a grid covering

the desired domain on the terrain are sampled from the set of patches. If this second task is performed on the GPU or on another computer, patches need to be transferred on the system bus or on a network. As we will show, queries are extremely efficient even using a single core, easily scaling up to hundreds of queries per second. The critical phases become transmission and tessellation. Our current GPU prototype yields interactive frame rates and we expect that an optimized GPU implementation, which will be the focus of our future work, will be able to obtain interactive frame rates on larger terrains with HD quality.

We present results produced using bilinear patches unless otherwise stated. Section 6.6 discusses performance when bicubic patches are used.

6.1 Pre-processing

The pre-processing computations executed by our system can be divided in three phases: mipmap generation, error evaluation and construction of the spatial index. Table 1 reports our preprocessing times for the full dataset, and for two scaled versions. Note that pre-processing is performed online, i.e. it is possible to add new data to a precomputed dataset without the need to rebuild it from scratch. This feature is unique of our method since, at the best of our knowledge, it is not available in any other work in the literature [12].

In our experiments, each patch covers a grid of 32x32 samples, while its kernel is made of the central 28x28 pixels.

Dataset		Preprocessing Time				Space overhead			
samples	size	Mipmap	Error	Index	Total	Mipmaps	Patches	Index	Total
1k × 1k	2M	0.1s	0.6s	0.05s	0.75s	702k	18k	12k	732k
4k × 4k	32M	0.9s	10s	0.83s	11.73s	11.2M	301k	202k	11.7M
16k × 16k	512M	12.6s	169s	14.6s	196.2s	179M	4.8M	3.2M	187M

Table 1. Time and space required to preprocess and store the multi-resolution model. From left to right: the time required to compute the mipmap, to evaluate the error associated with each patch and to build the spatial index; the space required to store the mipmaps, the patches and the spatial index.

The majority of time is spent on the first two phases, which would be simple to execute in parallel on multiple cores, unlike the last phase, which involves complex data structures.

6.2 Space overhead

On average, our multi-resolution model requires approximately 35% space more than the original dataset. A breakdown of the space occupied by the various components of our model is shown in Table 1. The majority of space is taken by the mipmap.

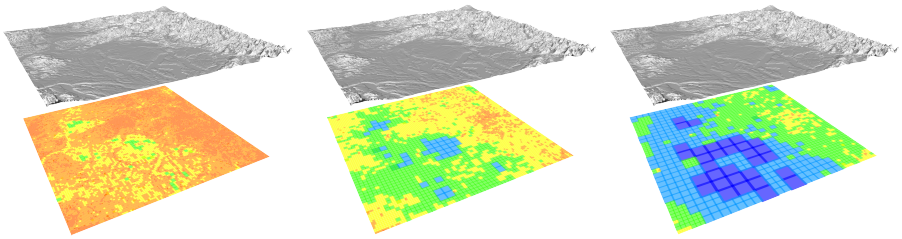


Fig. 5. Puget Sound Dataset (16k x 16k samples) rendered with error thresholds of 5, 20 and 50 meters. The colors on the bottom represent the size of the patch used to approximate the terrain. Blue and cyan corresponds to large patches, used to approximate flat zones, while red and orange indicates small patches required to represent fine details.

There is a trade-off between the space occupied by the multi-resolution model and the size of patches. Smaller patches increase adaptivity but take more space since they must be inserted and stored in the spatial index.

6.3 Spatial index queries

Uniform queries. Our system is able to execute 800 uniform queries per second with a 50m error. Queries with no error slow down the system to 55 queries per second. Note that the latter queries return the maximum number of patches at the highest level of detail possible.

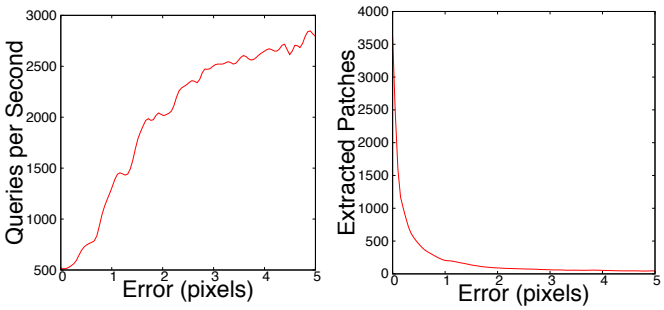
Figure 5 shows the results of three different queries performed with an error threshold of 5, 20 and 50 meters. Smaller patches are used to correctly represent fine details, while large patches are used in flat zones, even with a very low error threshold. High frequency detail is obviously lost as error increases.

View-dependent queries. A single view-dependent query representing a portion of terrain 15km long with an on-screen error of one pixel extracts approximately 250 patches and requires only 2.5ms. Thus, our system is able to query the spatial index at very high frame rates, meaning that the CPU time required by queries for every frame is negligible.

Figure 6 shows the number of view-dependent queries per second executed by our prototype and the number of extracted patches at different screen error thresholds. The use of progressive spatial queries could further increase performance.

6.4 Transmission of patches

To the GPU. As shown in Figure 7, sending all the extracted patches to the GPU takes a neglibile amount of time. Transmission of hundreds of patches (more than enough for high quality rendering) to the GPU requires less than a millisecond.



(a) (b)

Fig. 6. Number of queries per second (a) and number of extracted patches (b) while performing view-dependent queries at different screen error thresholds.

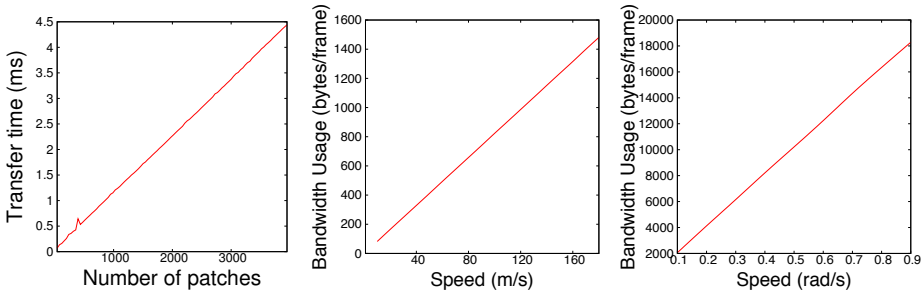
On a network. We have simulated the minimal traffic required to send patches on a network during a fly over the Puget Sound dataset at different speeds: only a few kb per frame are required to send the difference between two queries to the GPU (see (b) and (c) in Figure 7). Every patch that has to be sent to the GPU uses 4106 bytes, while the removal of a patch requires only to transfer its unique identifier (4 bytes). This makes our framework suitable for a client/server architecture since a reasonably low bandwidth is required to achieve high quility interactive rendering. Development of incremental queries would make it even more suitable.

6.5 GPU tessellation

Our GPU prototype already obtains interactive frame rates with a laptop-screen sized resampling grid (see Fig. 8). As expected, its performances scale linearly with the number of sampling points. We expect a future optimized implementation, as sketched in section 7, to easily reach interactive frame rates for HD resolutions.

6.6 Differences with bicubic patches

Changing type of patches influences differently the various steps of our framework. The preprocessing step is slowed down thirty times: this is due to the huge increase of the computational cost required for the evaluation of the bicubic patches. The construction of the spatial index is almost unaffected by the modification, since the only information that it needs is the maximal error associated with every patch. The space used is similar. The evaluation of the terrain is greatly slowed down. Currently, only a CPU implementation exists. While our current implementation can be used just for modeling purposes, a highly optimized GPU implementation would be required to reach interactive frame rates.



(a) (b) (c)

Fig. 7. (a) Time required to send a certain number of patches to the GPU on the system bus. (b) During a straight fly over terrain at different speeds, only a few bytes per frame must be sent through the network to update the set of patches to the viewpoint. (c) A rotation of the viewpoint requires slightly more bandwidth. Both (b) and (c) were performed with an allowed screen error of three pixels and every query extracted 70 patches on average, representing a portion of terrain 15km long.

7 Concluding Remarks

7.1 Benefits

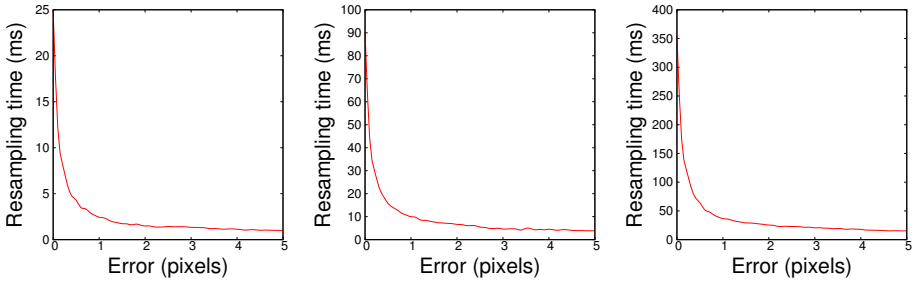
The main advantage of our method is the possibility to efficiently update the system with new heterogeneous grids, by automatically detecting and removing redundant data. Furthermore, our technique produces a multi-resolution C^k surface instead of a discrete model. The actual evaluation of the surface, which is the only computationally intensive task, can be demanded to the GPU, while keeping the communication between CPU and GPU limited. Texture and normal map can be easily integrated, since they can be associated to every patch and interpolated, with the same method used for the height values.

The space overhead is moderate, being approximately the same as the space used for a mipmap pyramid. The spatial index involves a negligible overhead, and it can be maintained in main memory even for huge terrain databases.

7.2 Limitations

Some limitations of our current approach come from the lack of certain theoretical bounds:

1. On the maximum number of patches that may overlap at a single point of terrain. In our experiments the number of overlapping patches never exceeded six, and it was four on average, but computation of Formula 1 at each sampling point may result computationally intensive even in this case.



(a) (b) (c)

Fig. 8. Time required to resample the terrain on the GPU using a 256×256 grid (a), a 512×512 grid (b) and a 1024×1024 grid (c) while performing view-dependent queries at different screen error thresholds.

2. On the accuracy of a sampled point. When evaluating the point, a patch that is as precise as required is surely present, given the spatial index query properties. Unfotunately, using Formula 1 the contribute of the most accurate patch could be slightly smoothed out by neighboring patches. In our experiments this effect was undetectable by human eye, but having a theoretical bound would be preferable.

Futhermore, our current GPU implementation is optimized neither for data transfer, nor for computational balance, as all data are transferred to the GPU at each frame, and all threads process all patches. We believe that a speedup of orders of magnitude could be gained with an optimized implementation, as outlined in the following.

7.3 Future work

Our current work is proceeding in several directions to extend our current model and implementation. Our aim is to overcome the limitations outlined in the previous section.

New blending function. We are exploring a blending function that will allow us to use only the most accurate patch for any point that falls within its kernel, while blending only at transitions between the extension zone of a more accurate patch and the kernel of a less accurate patch. This new method will allow us to improve accuracy and greatly speedup computation altogether, providing us with a theoretical bound on the error and performing blending only on very small (usually just two) and limited number of patches.

Optimized GPU tessellation. Computational load of GPU processors can be greatly improved by a proper distribution of patches to multiprocessors. In the

CUDA architecture, a fixed number (usually 16 or 32) of threads reside on the same processor core and share a fast memory between them that can be used as an explicit cache called "shared memory". These threads are said to belong to the same "warp" (see [10]). We are exploring a novel GPU parallel data structure that could exploit the fact that both sample points and patches share the same 2D geometrical domain. We may allocate sample points to threads in such a way that threads within a given warp map to neighboring samples. Then, we assign to the shared memory in the warp just the patches that contain such samples. This can be done in a pre-computation phase on the GPU, where each thread works on a patch; followed by an evaluation phase, where each thread evaluates a sample point. With this strategy, every thread works just on a subset of all patches, already cached inside the shared memory, thus greatly speeding up the tessellation phase.

Cache-aware and pre-fetching policies. Since our model can be useful in a variety of contexts - from real-time visualization on a local host, to client-server transmission on a geographic network - and it is especially relevant for huge databases, then the amount of data transferred between different levels of the memory hierarchy is of utmost importance in assessing its performance. Network bandwidth can be critical for a web application, as well as the bandwidth of system bus can be critical for CPU-GPU communication. I all such contexts, suitable policies can be developed to optimize performance in terms of trade-off between speed and quality. Fast compression/decompression mechanisms [18] can be adopted to compress patches or groups of patches (subgrids), thus reducing bandwidth usage. Suitable cache-aware policies can be also developed to decide, depending on both bandwidth and amount of memory available on the "client" side, the amount of data to be transferred and cached, and how to discard data from local memory when memory becomes scarce. Finally, for applications such as dynamic view-dependent rendering, suitable pre-fetching techniques can be developed to foresee the data needed to render the next frames ahead of time, and transfer data before they become necessary.

References

1. Lucas Ammann, Olivier Génevaux, and Jean-Michel Dischler. Hybrid rendering of dynamic heightfields using ray-casting and mesh rasterization. In *Proceedings of Graphics Interface 2010*, GI '10, pages 161–168, Toronto, Ont., Canada, Canada, 2010. Canadian Information Processing Society.
2. J. Bösch, P. Goswami, and R. Pajarola. Raster : simple and efficient terrain rendering on the gpu. In *Proceedings EUROGRAPHICS 2009 (Area papers)*, pages 35–42, Munich, Germany, 2009.
3. Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In Carl Gutwin and Stephen Mann, editors, *Graphics Interface*, pages 203–209. Canadian Human-Computer Communications Society, 2006.

4. Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Bdam - batched dynamic adaptive meshes for high performance terrain visualization. *Comput. Graph. Forum*, 22(3):505–514, 2003.
5. Christian Dick, Jens Krüger, and Rüdiger Westermann. GPU ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50, 2009.
6. Peter Lindstrom and Jonathan D. Cohen. On-the-fly decompression and rendering of multiresolution terrain. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 65–73, New York, NY, USA, 2010. ACM.
7. Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA, 1996. ACM.
8. Yotam Livny, Neta Sokolovsky, Tal Grinshpoun, and Jihad El-Sana. A gpu persistent grid mapping for terrain rendering. *Vis. Comput.*, 24(2):139–153, January 2008.
9. Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776, New York, NY, USA, 2004. ACM.
10. NVIDIA Corporation. NVIDIA CUDA C programming guide, 2012. Version 4.2.
11. Kyoungsu Oh, Hyunwoo Ki, and Cheol-Hi Lee. Pyramidal displacement mapping: a gpu based artifacts-free ray tracing through an image pyramid. In Mel Slater, Yoshifumi Kitamura, Ayellet Tal, Angelos Amditis, and Yiorgos Chrysanthou, editors, *VRST*, pages 75–82. ACM, 2006.
12. Renato Pajarola and Enrico Gobbetti. Survey of semi-regular multiresolution models for interactive terrain rendering. *Vis. Comput.*, 23(8):583–605, 2007.
13. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, September 2007.
14. Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
15. Art Tevs, Ivo Ihrke, and Hans-Peter Seidel. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In Eric Haines and Morgan McGuire, editors, *SI3D*, pages 183–190. ACM, 2008.
16. Marc Treib, Florian Reichl, Stefan Auer, and Rüdiger Westermann. Interactive editing of gigasample terrain fields. *Computer Graphics Forum (Proc. Eurographics)*, 31(2):383–392, 2012.
17. USGS and The University of Washington. Puget sound terrain. `http://www.cc.gatech.edu/projects/large_models/ps.html`.
18. Wladimir J. van der Laan, Andrei C. Jalba, and Jos B.T.M. Roerdink. Accelerating wavelet lifting on graphics hardware using cuda. *IEEE Transactions on Parallel and Distributed Systems*, 22:132–146, 2011.
19. Douglas Voorhies. Triangle-cube intersection. *Graphics Gems III*, pages 236–239, 1992.
20. K. Weiss and L. De Floriani. Simplex and diamond hierarchies: Models and applications. In H. Hauser and E. Reinhard, editors, *Eurographics 2010 - State of the Art Reports*, Norrköping, Sweden, 2010. Eurographics Association.

21. Holger Wendland. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in Computational Mathematics*, 4(1):389–396, 1995.