Interactive Curvature Tensor Visualization on Digital Surfaces*

Hélène Perrier[1], Jérémy Levallois[1,2], David Coeurjolly[1], Jean-Philippe Farrugia[1], Jean-Claude Iehl[1], and Jacques-Olivier Lachaud[2]

[1] Université de Lyon, CNRS
LIRIS, UMR5205, F-69622, France
[2] Université de Savoie Mont Blanc, CNRS
LAMA, UMR5127, F-73776, France

Abstract. Interactive visualization is a very convenient tool to explore complex scientific data or to try different parameter settings for a given processing algorithm. In this article, we present a tool to efficiently analyze the curvature tensor on the boundary of potentially large and dynamic digital objects (mean and Gaussian curvatures, principal curvatures, principal directions and normal vector field). More precisely, we combine a fully parallel pipeline on GPU to extract an adaptive triangulated isosurface of the digital object, with a curvature tensor estimation at each surface point based on integral invariants. Integral invariants being parametrized by a given ball radius, our proposal allows to explore interactively different radii and thus select the appropriate scale at which the computation is performed and visualized.

Keywords: Isosurface Visualization, Digital Geometry, Curvature Estimation, GPU.

1 Introduction

Volumetric objects are being more and more popular in many applications ranging from object modeling and rendering in Computer Graphics to geometry processing in Medical Imaging or Material Sciences. When considering large volumetric data, interactive visualization of those objects (or isosurfaces) is a complex problem. Such issues become even more difficult when dynamic volumetric datasets are considered. Beside visualization, we are also interested in performing geometry processing on the digital object and to explore different parameter settings of the geometry processing tool. Here, we focus on curvature tensor estimation (mean/Gaussian curvature, principal curvatures directions...). Most curvature estimators require a parameter fixing the scale at which the computation is performed. For short, such parameter (integration radius, convolution

* This work has been mainly funded by DigitalSnow ANR-11-BS02-009, KIDICO ANR-2010-BLAN-0205 and PRIMES Labex ANR-11-LABX-0063/ ANR-11-IDEX-0007 research grants.

kernel size...) specifies a scale for analyzing the object surface, and is naturally related to the amount allowed perturbations on input data. As a consequence, when using such estimators, exploring different values of this parameter is mandatory. However, this is usually an offline process, due to the amount of computations that need to be done.

Contributions In this work, we propose a framework to perform interactive visualization of complex 3D digital structures combined with a dynamic curvature tensor estimation. We define a fully data parallel process on the GPU (Graphics Processor Unit) to both efficiently extract adaptive isosurface and compute per vertex curvature tensor using Integral Invariants estimators. This system allows us to visualize curvature tensor in real-time on large dynamic objects. Our approach combines a GPU implementation of pointerless octrees to represent the data, an adaptive viewpoint-dependent mesh extraction, and a GPU implementation of integral invariant curvature estimators.

Related works Extracting and visualizing isosurface on volumetric data has been widely investigated since the seminal Marching Cubes approach by Lorensen and Cline [11]. This approach being data parallel, GPU implementation of this method is very efficient [16]. However, such technique generates a lot of triangles which is not well suited to large digital data. Hence, adaptive approaches have been proposed in order to optimize the triangulated mesh resolution according to the object geometry or the viewpoint. In this topic, many solutions have been developed in Computer Graphics [15, 14, 6, 9, 10]. The method developed by Lengyel *et al.* [6] suits best our needs. It combines an octree space partitioning with new Marching Cubes configurations to generate adaptive meshes on CPU from static or near static data. We propose here a full GPU pipeline inspired by this method, that maintains a view dependent triangulation. Our high framerate allows us to inspect dynamic 3D data in real-time.

Curvature estimation on discrete or digital surface has also been widely investigated. In [2], authors propose digital versions of Integral Invariant estimators [13, 12] in order to estimate the complete curvature tensor (mean/Gaussian curvatures, principal curvatures, principal curvature directions, normal vector field) on digital surfaces. Such approaches are based on an integration principle using a ball kernel of a given radius. Additionally, authors have demonstrated that these estimators have multigrid convergence properties. In this article, we also propose an efficient GPU implementation of such estimators to visualize such curvature fields in real-time and to interactively change the value of the kernel radius.

In this paper, we first present the previous works on curvature tensor estimation. Then, we propose an efficient GPU approach to extract a triangulated isosurface from a digital object. Finally, we propose a fully-parallel GPU pipeline to compute curvature tensor in real-time and present our results.

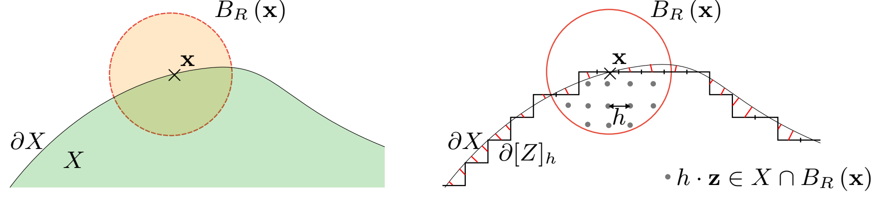

Fig. 1. Integral invariant computation (*left*) and notations (*right*) in dimension 2 [2].

2 Curvature Tensor Estimation

In our context, we consider digital shapes (any subset Z of $\mathbb{Z}^d$) and boundaries of digital shapes $Bd(Z)$. We denote by $\mathtt{G}_h(X)$ the Gauss digitization of a shape $X \subset \mathbb{R}^d$ in a $d-$dimensional grid with grid step h, *i.e.* $\mathtt{G}_h(X) := \{\mathbf{z} \in \mathbb{Z}^d,\ h \cdot \mathbf{z} \in X\}$. For such digitized set $Z := \mathtt{G}_h(X)$, $[Z]_h$ is a subset of $\mathbb{R}^d$ corresponding to the union of hypercubes centered at $h \cdot \mathbf{z}$ for $\mathbf{z} \in Z$ with edge length h. By doing so, both ∂X and $\partial[Z]_h$ are topological boundaries of objects lying in the same space (see Fig. 1-b). Note that the combinatorial digital boundary $Bd(Z)$ of Z made of cells in a cellular Cartesian complex (*pointels*, *linels*, *surfels*, ...), can be trivially embedded into $\mathbb{R}^d$ such that it coincides with $\partial[Z]_h$.

In [1], authors define a 2D digital curvature estimator $\hat{\kappa}^R$ and a 3D digital mean curvature estimator $\hat{H}^R$ based on the digital volume estimator $\widehat{\mathrm{Vol}}(Y,h) := h^d \mathrm{Card}(Y)$ (area estimator $\widehat{\mathrm{Area}}(Y,h)$ in dimension 2):

Definition 1 *Given the Gauss digitization $Z := \mathtt{G}_h(X)$ of a shape $X \subset \mathbb{R}^2$ (or $\mathbb{R}^3$ for the 3D mean curvature estimator),* digital curvature estimators *are defined for any point $\mathbf{x} \in \mathbb{R}^2$ (or $\mathbb{R}^3$) as:*

$$\forall 0 < h < R, \quad \hat{\kappa}^R(Z,\mathbf{x},h) := \frac{3\pi}{2R} - \frac{3\widehat{\mathrm{Area}}(B_{R/h}(\mathbf{x}/h) \cap Z, h)}{R^3}, \tag{1}$$

$$\hat{H}^R(Z,\mathbf{x},h) := \frac{8}{3R} - \frac{4\widehat{\mathrm{Vol}}(B_{R/h}(\mathbf{x}/h) \cap Z, h)}{\pi R^4}. \tag{2}$$

where $B_{R/h}(\mathbf{x}/h)$ is the ball of digital radius R/h centered on digital point $(\mathbf{x}/h) \in Z$.

Such curvature estimators have multigrid convergence properties [1]: when the digital object becomes finer and finer, *i.e.* when the digitization step h tends to zero, the estimated quantities on $\partial[\mathtt{G}_h(X)]_h$ converges (theoretically and experimentally) to the associated one on ∂X in $O\left(h^{\frac{1}{3}}\right)$ for convex shapes with at least C^3-boundary and bounded curvature (setting $R := kh^{\frac{1}{3}}$ for some $k \in \mathbb{R}$).

In [2], authors have also defined 3D digital principal curvature estimators $\hat{\kappa}_1^R$ and $\hat{\kappa}_2^R$ on $Z \subset \mathbb{Z}^3$ based on digital moments:

Definition 2 *Given the Gauss digitization* $Z := \mathtt{G}_h(X)$ *of a shape* $X \subset \mathbb{R}^3$*,* 3D digital principal curvature estimators *are defined for any point* $\mathbf{x} \in \mathbb{R}^3$ *as:*

$$\forall 0 < h < R, \quad \hat{\kappa}_1^R(Z, \mathbf{x}, h) := \frac{6}{\pi R^6}(\hat{\lambda}_2 - 3\hat{\lambda}_1) + \frac{8}{5R}, \tag{3}$$

$$\hat{\kappa}_2^R(Z, \mathbf{x}, h) := \frac{6}{\pi R^6}(\hat{\lambda}_1 - 3\hat{\lambda}_2) + \frac{8}{5R}, \tag{4}$$

where $\hat{\lambda}_1$ *and* $\hat{\lambda}_2$ *are the two greatest eigenvalues of the covariance matrix of* $B_{R/h}(\mathbf{x}/h) \cap Z$*.*

The covariance matrix needs to compute digital moments of order 0, 1 and 2 (see Equation 20 of [2] for more details). These estimators are proven convergent in $O\left(h^{\frac{1}{3}}\right)$ when setting the ball radius h in $R = kh^{\frac{1}{3}}$, where k is a constant related to the maximal curvature of the shape, on convex shapes with at least C^3-boundary and bounded curvature [2]. Additionally, eigenvectors associated to $\hat{\lambda}_1$ and $\hat{\lambda}_2$ of the covariance matrix are principal curvature direction estimators $\hat{\mathbf{w}}_1^R$ and $\hat{\mathbf{w}}_2^R$. The smallest eigenvector corresponds to the normal direction $\hat{\mathbf{n}}^R$ at $\mathbf{x}$. Convergence results can be found in [5].

It has been shown that the radius of the ball depends on the geometry of the underlying shape. In [7], a proposal was made for a parameter-free estimation of the radius of the ball by analyzing the shape *w.r.t.* the local shape geometry using maximal digital straight segments of the digital boundary. In [8], these estimators have been analyzed in scale-space (for a range of radii) for a given digital shape. This allows to detect features of the shape thanks to the behavior of estimators on singularities. As a consequence, for all these integral invariant based approaches, we need to consider different ball radius which could be time consuming when implemented on CPU. We propose here a fully parallel implementation on GPU allowing us to change the radius and thus update the estimated quantities in real-time.

3 Isosurface Extraction on GPU

In this section, we detail the adaptive isosurface extraction algorithm. The proposed approach uses an octree representation of the input object on which an adaptive Marching Cube builds the isosurface efficiently. Such hierarchical representation of the object allows us to handle large datasets and to locally adapt the level of details *w.r.t.* the geometry or camera position. We first present the octree representation and then the isosurface extraction.

3.1 Linear Octree Representation

Representing a hierarchical structure on GPU is usually challenging since such a data parallel component is unable to handle recursivity. Efficient spatial tree encoding can be achieved using pointerless structures such as linear quadtrees or octrees Gargantini [4]. This structure indexes each cell by a *Morton code*:

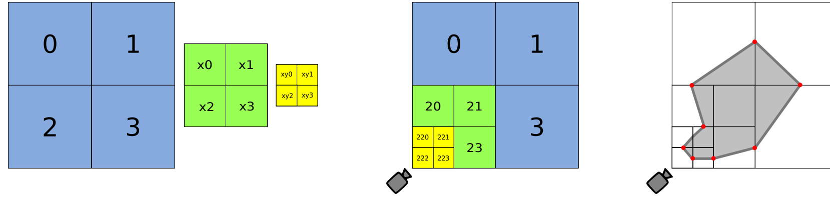

Fig. 2. Morton codes associated to cells of a linear quatree. Each morton code of a child cell is obtained by adding a suffix to its parent code *(left)*. The adaptive representation consists of quadtree cells whose depth is view point dependent *(middle)*. Finally, adaptive Marching Cubes is used to generate the triangulation *(right)*.

the code of children cells are defined by the code of the parent suffixed by two bits (in dimension 2, three bits in dimension 3) (see Figure 2-*left*). A cell's code encodes its position *w.r.t.* its parent cell and its complete path to the tree root. Hence, the tree is fully represented as a linear vector of its leaves. Furthermore, cell operations such as subdivision, merging can be efficiently implemented using bitwise operations on the Morton code. In the following, we use the GPU friendly implementation proposed by Dupuy *et al.* [3].

3.2 Data parallel and adaptive mesh generation

Using this spatial data structure, a triangulated mesh can be constructed using Marching Cubes [11] (MC for short): the triangulation is generated from local triangle patches computed on local cell configurations. Such approach is fully parallel and easy to implement on the GPU. However, since adjacent cells may not have the same depth in the octree, original Lorensen and Cline's rules need to be updated (see Figure 2-*right*). Many authors have addressed this problem both for primal and dual meshes [15, 14, 6, 9, 10].

In the following, we use the algorithm proposed by Lengyel *et al.* [6]. First, this approach forces the octree structure to make sure that the depth difference between any two adjacent cells is at most one. Then, Lengyel *et al.* introduce the concept of transition cells. Those cells are defined to be inserted between two neighboring octree cells of different depth. With specific MC configurations for their triangulation, a crack free mesh can be extracted.

Similarly to original MC algorithm, this approach is well suited to a GPU implementation: given a set of cells (a vector of morton codes), each triangle patch can be extracted in parallel for both regular cells and transition cells.

3.3 Level of Details Criteria and Temporal Updates

As illustrated in Figure 2-*middle*, we propose a viewpoint dependent criterion to decide if a cell needs to be refined: the closer we are to the camera, the finer the cells are. Such a criterion is also well suited to GPU since it can be evaluated independently on every cell. Figure 3 illustrates our level of details (LoD for short) criterion. In dimension 2, if α denotes the viewing angle, an object at a

distance d from the camera has a projected size on screen of $2 \cdot d \cdot \tan(\alpha)$. (see Figure 3-*left*). Our distance criterion is based on the ratio (*visibility ratio* in the following) between the cell diameter $l(c)$ (power of 2 depending on the depth), and its projected size. For a given cell c, split and merge decision are based on this visibility ratio:

- c is split if its children cells have a visibility ratio greater than constant k ;
- c and its sibling cells are merged if their parent cell c' has a visibility ratio lower than k ;
- otherwise, the cell c stays for the next frame.

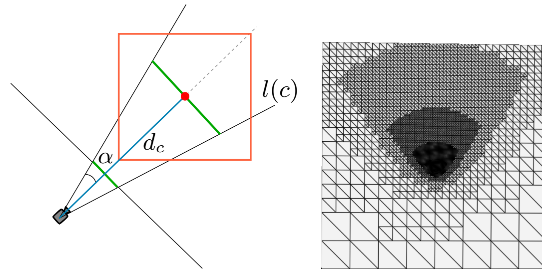

Fig. 3. Notations *(left)* and adaptive meshing in dimension 2 using the LoD distance and angular criterion *(right)*.

Using such a criterion, split and merge decisions are computed in parallel from the morton codes of all cells.

Once decisions have been made, a new set of cells is sent to the mesh generation step described above. Finally, before constructing the triangulation from remaining cells and transition cells, geometrical culling is performed in order to skip the triangle patch construction for cells that are not visible. Figure 4 illustrates the overall fully data parallel pipeline.

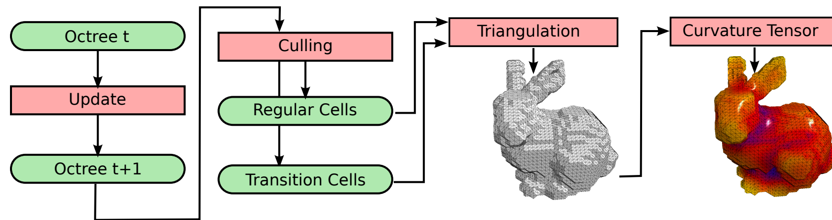

Fig. 4. GPU pipeline summary. Data buffers are represented in green and computations in red. Each computation retrieves data from a buffer and fills a new one.

4 Interactive Curvature Computation on GPU

We first present design principles for the GPU implementation and then present our approaches. The general idea is to perform an Integral Invariant computation on GPU at each vertex of the generated triangulated mesh. Since GPU have massively parallel architectures, we can do all those computations in parallel to obtain a very efficient implementation. Please note that when the LoD criterion is removed, each MC vertex is exactly centered at a surfel center. At different depth of the octree, MC vertices still correspond to surfel centers of subsampled versions of the input object. Hence, Integral Invariant framework defined in Section 2 is consistent with the triangulated surface obtained on GPU: triangles are used for visualization purposes but all computations are performed on the digital object $\mathtt{G}_h(X)$ for estimators (1), (2), (3) and (4).

To implement the integration on $B_R(\mathbf{x}) \cap X$ (Fig. 1), several strategies have been evaluated.

4.1 Per Vertex Real-time Computation on GPU

Naive Approach. A first simple solution consists in scanning all digital points, at a given resolution, lying inside the integration domain (Fig. 5-$left$) and then estimating the geometrical moment as the sum of the geometrical moments of each elementary cubes lying in the intersection (see Eq. (1) to (4)). This is exactly similar to what is done on the CPU. On the GPU, we can exploit *mipmap* textures to obtain multi-resolution information. If the input binary object is stored in a 3D texture, GPU hardware constructs a multi-resolution pyramid (mipmap) such that 8 neighboring voxel intensities at level l are averaged to define the voxel value at level $l+1$. If the level 0 corresponds to the binary input object, at a given level l, a texture probe at a point (x, y, z) returns the fraction of $\mathtt{G}_h(X)$ belonging to the cube of center (x, y, z) and edge length 2^l. As a consequence, we can approximate the volume of $B_R(\mathbf{x}) \cap X$ by considering mipmap values at a given resolution l (Fig. 5-$right$). In this case, errors only occurs for cells lying inside X (with density 1) not entirely covered by $B_R(\mathbf{x})$. Furthermore, the *mipmap* texture can be used to design, using a single texture probe, a fast inclusion test of a given cell c at level l into the shapes: we say that c is in X if its density (retrieved from the texture probe at level l) is greater than 1/2. The idea here is to mimic a kind of adaptive Gauss digitization process.

Hierarchical Decomposition. Using the hierarchical nature of the mipmap texture, we could also consider hierarchical decompositions of $B_R(\mathbf{x})$. The idea is to decompose the ball into mipmap cells of different resolution in order to limit the number of texture access (important bottleneck on GPU hardwares) and to get better approximated quantities. On the GPU, computing a hierarchical decomposition of $B_R(\mathbf{x})$ at each point $\mathbf{x}$ is highly inefficient since the hardware optimizes the parallelism only when the micro-program described in the shader has a predictive execution flow. Hence, implementing the recursive algorithm (or its de-recursified version) requires a lot of branches (conditional

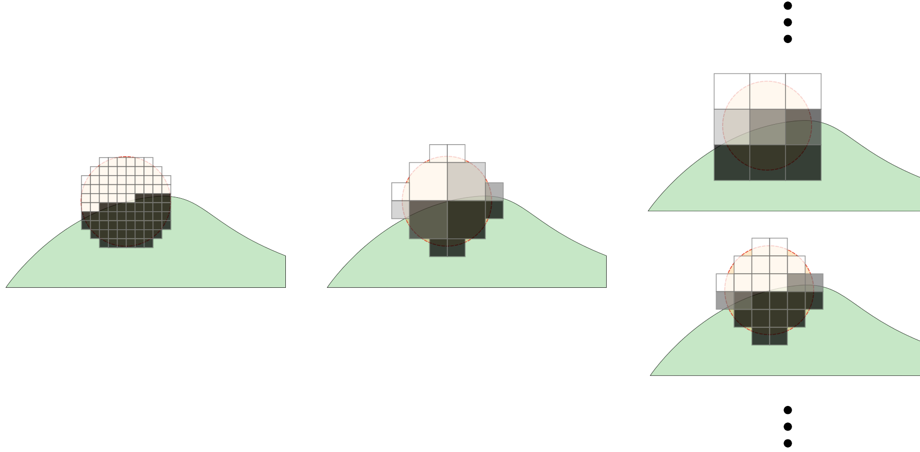

Fig. 5. Here are the three approaches we used to evaluate the integrand. (*left*) is the naive approach, where we sum all the digital points that are inside the integrand domain (*middle*) shows the hierarchical decomposition of the ball, thus limiting the number of texture probes to do. (*right*) illustrates the dynamic approach, we probe the textures at a higher resolution and we refine it at each frame until we reach the finer level.

if tests) in the flow. We have thus considered a fast multi-resolution approximation as illustrated in Fig. 5-*middle*: for a given radius R, we precompute a hierarchical decomposition of B_R. Such decomposition is made of octree cells at different resolutions. At a given MC vertex $\mathbf{x}$, the algorithm becomes simple since we just scan the ball cells and estimate the area (or other moments) from the mipmap values associated to this cell. Note that these precomputed cells in the B_R decomposition may not be aligned with mipmap cells. However, we can use GPU which interpolates mipmap values if we probe at non-discrete positions. For a given radius R, the expected number of cells in B_R is in $O(\log R)$. Implementation details on the hierarchical octree representation can be found in the supplementary material (see Section 6).

Dynamic Refinement. In this last approach, we want to optimize the interactivity and the execution flow or parallelism on GPU. The integration is simply computed using a regular grid at different resolution l (Fig. 5-$right$). The shader code becomes trivial (simple triple loop) and a lot of texture fetches are involved, but interactivity can be easily obtained. Indeed, we consider the following multi-pass approach:

1. When the surface geometry has been computed, we set the current level l at an high level $l{:=}l_{\max}$ of the mipmap pyramid.
2. We compute the integrals (and the curvature tensor) at the mipmap level l and send the estimated quantities to the visualization stage.
3. If the user changes the camera position or the computation parameters, we return to step 1.

4. If the current framerate is above a given threshold, we decrease l and return to step 2 if the final l_0 level has not been reached yet.

A consequence of the multi-pass approach is that when there is no interaction (camera settings, parameters), the GPU automatically refines the estimation. Even if step 2 is quite expensive, in $O\left(\left(\frac{R}{2^l}\right)^3\right)$, the interactive control in step 3 considerably improves the interactive exploration of the tensor with fast preliminary approximations which are quickly refined.

Compared to the hierarchical approach, no precomputation is required for a given radius R. As a consequence, we could even locally adapt the ball radius to the geometry (for instance following the octree depth of the current MC vertex). In the next section, we evaluate the performances of both approaches.

5 Experiments

5.1 Full resolution experiment

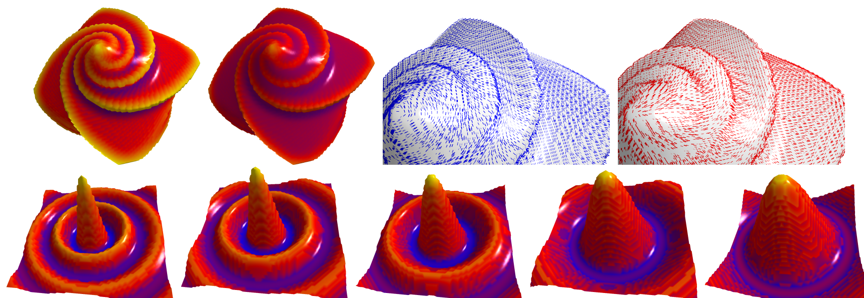

Fig. 6. *First row*: Mean and Gaussian curvature estimation, (zoom of) first and second principal directions estimation on "OctaFlower" with a digital domain of 130^3. *Second row:* Mean curvature computed in real-time (around 20 FPS) on a dynamic object.

We first evaluate curvature estimations on a full-resolution geometry obtained by disabling the LoD criterion at the mesh generation step. Figure 6-top shows results of curvature tensor estimation (mean, Gaussian, first and second principal directions) on "OctaFlower" at level l_0 (considered as our ground truth in the following). Figure 6-bottom shows mean curvature estimation in real-time on a dynamic object. For this case, we simply evaluate the implicit expression at each vertex of a cell to decide if such cell is included or not into $B_R(\mathbf{x}) \cap X$ instead of updating the *mipmap texture* of densities at each time step. For all illustrations, we use directly normal vectors computed by algorithm discussed in Section 2.

Then, we compare the approximations made by computing curvature from level $l \geq l_0$ in Figure 7. We can see that results using approximation seems to quickly converge to ground truth results. Table 1 shows numerical results of L_∞ error (maximal absolute difference for all vertices) and L_2 error (mean

squared errors of all vertices), as well the number of *mipmap texture* fetches. We can also see that the number of texel fetch, required to compute the curvature, reduces drastically when computing approximations. However, at higher levels, approximation errors become more important. Those levels are thus never used in practice, they are presented here to illustrate how the refining process converges to the l_0 level.

We also compare them with the hierarchical algorithm (as discussed in Section 4.1). This method introduces a higher error when compared with l_0. This is due to the precomputation of the subdivision that no longer ensures to fetch data at the center of a mipmap cell. The GPU interpolation reduces the bias, but it does not remove it.

5.2 Fully adaptive evaluation

When dealing with large datasets, we cannot expect real-time curvature tensor estimation if we consider the full resolution geometry, due to the huge amount of data to process. By computing a dynamically refined approximate curvature tensor estimation joined with an adaptive triangulation (as discussed in Section 4.1), we manage to maintain a real-time framerate by giving control over the amount of data to process at each frame. In Figure 8, we compare timings (in logarithmic scale) for a triangulation that is dynamically refined according to its distance to the camera. We measured those timings using the multiresolution regular and the hierarchical algorithm.

First we can note that the required time to compute the ground truth curvature for an object is usually as high as the time required to extract its geometry and to compute all of the above levels. Using approximations is thus mandatory. Another advantage with approximations is that it allows us to get a visualization of our object as soon as we run the application. This allows for real-time interactions with the object, required in order to change the visualized quantity, curvature radius, etc.

It is also visible in Figure 8 that a hierarchical decomposition greatly reduces the curvature computation time, especially with big radii. However, due to the precomputation and the current hierarchical structure, this algorithm is biased and creates an error (presented in Table 1) that needs to be considered.
Figure 9 shows curvature computation and exploration in real-time on large

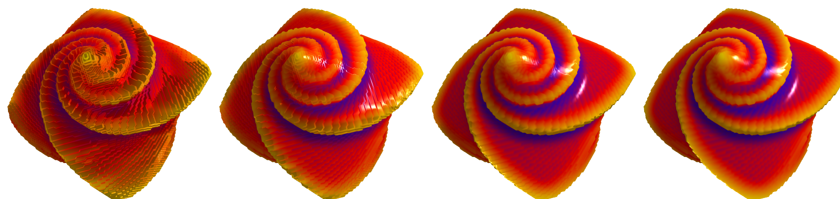

Fig. 7. Illustration of mean curvature computation on "OctaFlower" (digital domain of 130^3) using mipmap approximation with different levels: l_3, l_2, l_1 and l_0 (*i.e.* no approximation).

		H	l_4	l_3	l_2	l_1	l_0
Number of	$R = 8$	1468		2	28	260	2104
texture fetches	$R = 16$	5706	2	28	90	2120	17080
L_∞ error	$R = 8$	0.051		0.306	0.085	0.047	0
($w.r.t.\ l_0$)	$R = 16$	0.053	0.146	0.041	0.017	0.005	0
L_2 error	$R = 8$	3.51e-05		2.67e-04	7.57e-05	2.02e-05	0
($w.r.t.\ l_0$)	$R = 16$	4.43e-05	1.39e-04	4.16e-05	1.57e-05	2.07e-06	0

Table 1. Comparison of number of texture fetches, L_2 and L_∞ error obtained on "OctaFlower" with a digital domain of 130^3 when computing mean curvature with two radii: 8 and 16, with hierarchical algorithm (H) and $l \geq l_0$ approximation algorithms. The object is triangulated at full resolution with a regular grid and contains 282,396 vertices.

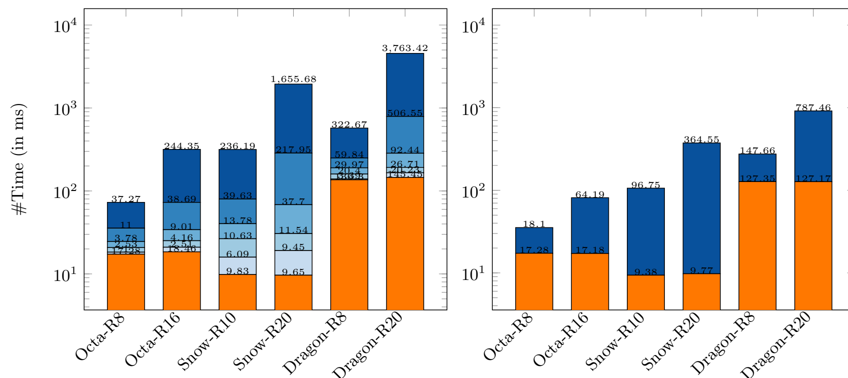

Fig. 8. Timings in milliseconds (in logscale) obtained while visualizing an adaptive triangulation on three objects – "OctaFlower" (digital domain of 130^3), "Snow microstructures" (233^3) and "XYZ-Dragon" (510^3) – by computing the curvature with a regular grid (*left*) and with hierarchical algorithm (*right*), with two different radii for each object. In *orange color*: time required to extract the triangulation. In *blue color*: time required to compute the curvature tensor at different levels: from l_4 (light blue) to l_0 (dark blue). Timings are given using a NVIDIA GeForce GTX 850M GPU.

datasets: "XYZ-Dragon" (with digital domain of 512^3) and "Snow microstructures" (233^3).

6 Conclusion and Discussion

In this article, we have proposed a fully data parallel framework on GPU hardware which combines an adaptive isosurface construction from digital data with a curvature tensor estimation at each vertex. Using this approach, we can explore in real-time different curvature measurements (mean, Gaussian, principal

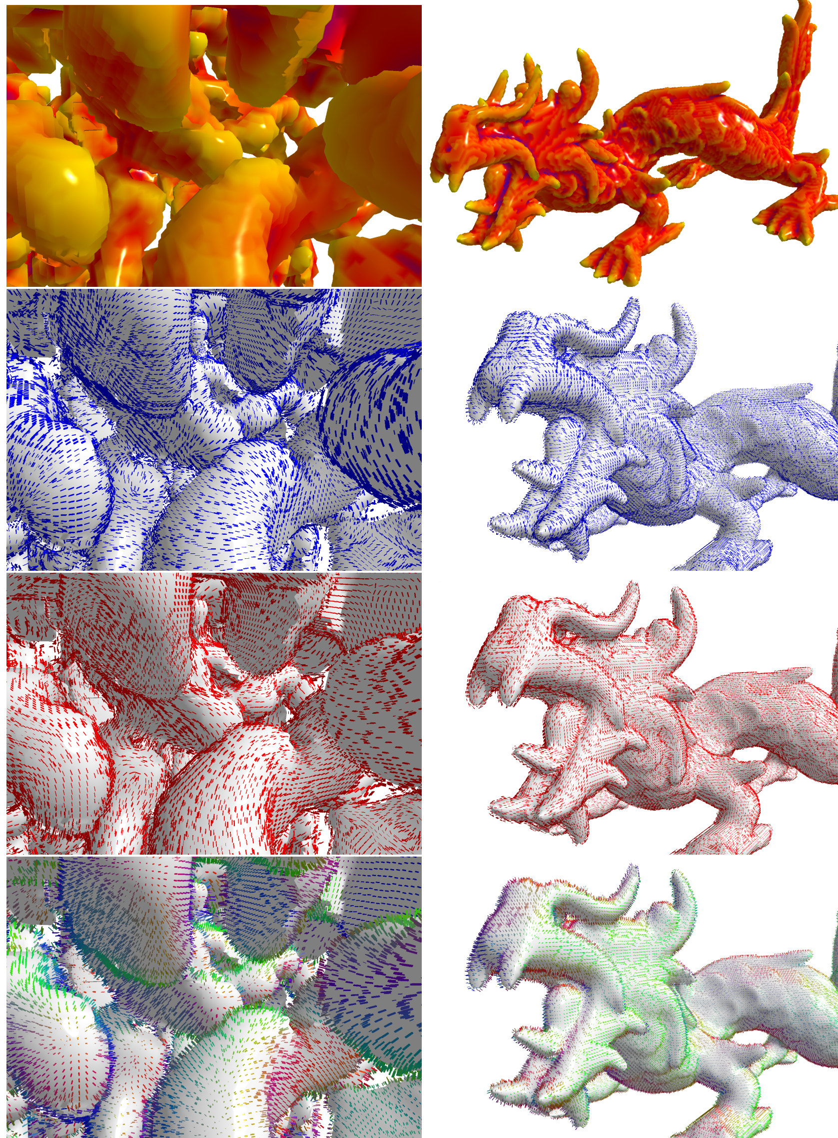

Fig. 9. *Left column*: Mean curvature, first and second principal directions and normal vector field estimation on “Snow microstructures” (233^3 and $R = 8$). *Right column:* Mean curvature, first and second principal directions and normal vector field estimation on “XYZ-Dragon” (510^3 and $R = 8$). Normal vectors are colored with a mapping of their component to RGB color space.

directions) with different ball radii on potentially large dynamic dataset. Our proposal relies on both a linear octree representation with Morton codes and an efficient integral computation on GPU. The source code and additional material (video, ...) are available on the project website (`https://github.com/dcoeurjo/ICTV`).

References

1. Coeurjolly, D., Lachaud, J.O., Levallois, J.: Integral based curvature estimators in digital geometry. In: Discrete Geometry for Computer Imagery. pp. 215–227. Springer (2013)
2. Coeurjolly, D., Lachaud, J.O., Levallois, J.: Multigrid convergent principal curvature estimators in digital geometry. Computer Vision and Image Understanding 129, 27–41 (2014)
3. Dupuy, J., Iehl, J.C., Poulin, P.: GPU Pro 5, chap. Quadtrees on the GPU. A K Peters/CRC Press (Mar 2014), `http://liris.cnrs.fr/publis/?id=6299`
4. Gargantini, I.: An effective way to represent quadtrees. Communications of the ACM 25(12), 905–910 (1982)
5. Lachaud, J.O., Coeurjolly, D., Levallois, J.: Robust and convergent curvature and normal estimators with digital integral invariants. In: Modern Approaches to Discrete Curvature. Lecture Notes in Mathematics, Springer International Publishing (2016, forthcoming)
6. Lengyel, E.S., Owens, J.D.: Voxel-based terrain for real-time virtual simulations. University of California at Davis (2010)
7. Levallois, J., Coeurjolly, D., Lachaud, J.O.: Parameter-free and multigrid convergent digital curvature estimators. In: Discrete Geometry for Computer Imagery. pp. 162–175. Springer (2014)
8. Levallois, J., Coeurjolly, D., Lachaud, J.O.: Scale-space feature extraction on digital surfaces. Computers and Graphics p. 12 (2015)
9. Lewiner, T., Mello, V., Peixoto, A., Pesco, S., Lopes, H.: Fast generation of pointerless octree duals. Comput. Graph. Forum 29(5), 1661–1669 (2010), `http://dx.doi.org/10.1111/j.1467-8659.2010.01775.x`
10. Lobello, R.U., Dupont, F., Denis, F.: Out-of-core adaptive iso-surface extraction from binary volume data. Graphical Models 76(6), 593–608 (2014), `http://dx.doi.org/10.1016/j.gmod.2014.06.001`
11. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. ACM Computer Graphics 21(4) (1987)
12. Pottmann, H., Wallner, J., Huang, Q., Yang, Y.: Integral invariants for robust geometry processing. Computer Aided Geometric Design 26(1), 37–60 (2009)
13. Pottmann, H., Wallner, J., Yang, Y., Lai, Y., Hu, S.: Principal curvatures from the integral invariant viewpoint. Computer Aided Geometric Design 24(8-9), 428–442 (2007)
14. Schaefer, S., Warren, J.: Dual marching cubes: Primal contouring of dual grids. In: Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on. pp. 70–76. IEEE (2004)
15. Shu, R., Zhou, C., Kankanhalli, M.S.: Adaptive marching cubes. The Visual Computer 11(4), 202–217 (1995)
16. Tatarchuk, N., Shopf, J., DeCoro, C.: Real-time isosurface extraction using the gpu programmable geometry pipeline. In: ACM SIGGRAPH 2007 courses. pp. 122–137. ACM (2007)