Eurographics Symposium on Point-Based Graphics (2006)
M. Botsch, B. Chen (Editors)

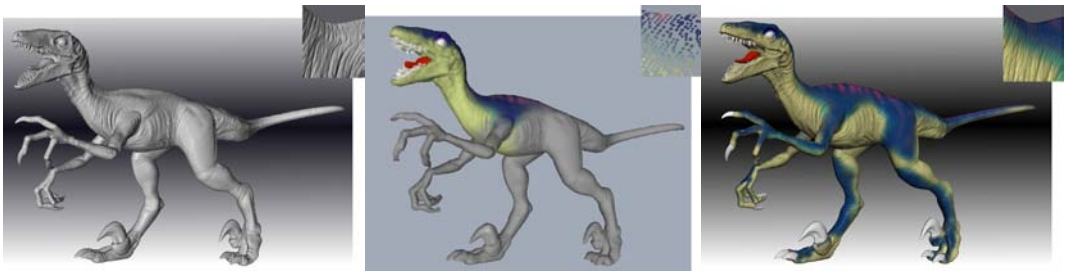
Interactive Out-Of-Core Texturing with Point-Sampled Textures

Tamy Boubekeur Christophe Schlick

LaBRI - INRIA - CNRS - University of Bordeaux

Abstract

The visualization of huge 3D objects becomes available on common workstations thanks to highly optimized data-structures and out-of-core frameworks for rendering. However, the editing, and in particular, the texturing of such objects is still a challenging task, since usual methods for optimized rendering are not easily amenable to interactive modification. In this paper, we introduce the idea of point-sampled textures, and show how to interactively texture such a huge model at various scales, without any parameterization. An adaptive in-core point-based approximated geometry is first created by employing an efficient out-of-core point-sampling algorithm. This simplified geometry is then used for an interactive and multi-scale point-based texturing. Finally, a feature-preserving kernel is used to convert the point-based model into a global 3D texture which can be applied back on the initial huge geometry. Our technique thus provides a flexible tool to generate, edit and apply size-independent textures to a wide range of huge 3D objects thanks to point-based methods.



Example: *Starting from a large (8M polygons here) out-of-core geometry (left), our technique generates an adaptive in-core point-based approximated geometry (150k samples) that is used for interactive multi-scale texturing (middle). The so-defined "point-sampled texture" is then upscaled with a feature-preserving kernel, and applied on the initial geometry (right).*

1. Introduction

The interactive texturing of 3D objects is a key step in the editing of the final object appearance in computer graphics productions. As usual with interactive tools, the size of the in-core model must be kept low since the dynamic information added during the interactive editing process would break any highly-optimized data-structures, from on-GPU vertex buffer objects to out-of-core representations of large objects.

In recent years, meshes of hundred millions of polygons have become quite common, with the achievement of high-quality 3D acquisition devices. So it seems quite natural for the user to be able to edit the appearance (diffuse color, specular value, etc) of the full resolution model, while preserving the small geometric features accurately captured by laser range scanners. Note that even if some scanners provide the color information during the scanning process, this information can rarely be directly exploited since the captured appearance strongly depends on the lighting conditions. Moreover, the user may want to define different surface attributes than the acquired colors or to add some informative features (e.g. try to focus attention on a specific part of an object). For this purpose, the need for an interactive texturing technique able to manage gigantic objects has grown and this was the motivation of the work presented here.

Our approach at a glance: The goal we try to reach is to interactively texture objects that are too huge to provide an interactive framerate when loaded and edited directly. We propose a multi-scale framework, illustrated on Figure 1, essentially composed of three steps:

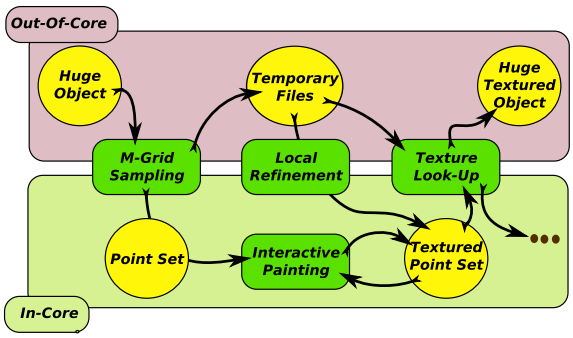


Figure 1: *Framework for out-of-core interactive texturing.*

1. **Out-of-core point-sampling**: by performing an out-of-core point-sampling of an initial huge object, we are able to represent arbitrary topology from various kinds of surface definition (polygonal soups, indexed meshes, unorganized point clouds), without dealing with any local structure-preserving operator, usual in mesh representations
2. **Interactive texturing**: the attributes of the so-defined in-core point cloud are then edited with usual point-based editing tools; this simplified object can be locally refined (from the original large geometry or from the brush) for specific complex features, taking benefit of the easy multi-scale insertion of samples in a point cloud
3. **Point Sampled Texture**: at any time, the in-core textured point set can be casted to a *point-sampled texture* (PST) and either be applied to the original large object through an out-of-core streaming process or directly used for per-pixel look-up at rendering time, such as with ray-tracing.

For the sake of simplicity, we will essentially discuss here the construction of one single color texture. But, as usual with texturing tools, complex texture may be built incrementally by assigning different textures for different material channels, to get more complex shading (appearance composition with specular, ambient, emissive and/or diffuse textures, see Figure 8).

The main choice we have done with this framework is to consider *volumetric textures*. Using 2D textures would have required global or piecewise parameterization, a process far from being simple in the case of large objects. Moreover, since we propose a multi-scale approach, samples are often inserted in the in-core point set during the editing, which would make the construction of a consistent parameterization even more complicated. In such a case, dealing with 3D textures makes it straightforward to compute the color of a given point, even if it does not lie exactly on the in-core surface. Furthermore, 3D textures easily handle procedural textures, such as Perlin or Wavelet noise [Per85, CD05], which offers a large variety of additional effects.

Previous Work: Direct interactive texturing of 3D objects has been an issue in computer graphics for many years. One of the first complete framework for interactive 3D painting was the WYSIWYG painting tool of Hanrahan and Haeberti [HH90]. Their system allows the user to interactively paint colors and materials directly on a 3D model, introducing a simple brush metaphor. The authors were yet pointing the usefulness of such a system for 3D scanned models.

Recently, the idea that 3D textures could be an interesting alternative to usual 2D textures in a painting tool has been independently developed by DeBry et al. [gDGPR02] and Benson and Davis [BD02] who introduced the idea of *octree textures*. The main idea is to set a per-node color at each level of the octree hierarchy and use it to color an object embedded in its volume. Note that octree textures may be interactively constructed or sampled from an existing texture [LHN05], without requiring any parameterization. Another great advantage of octree textures is their local control, which is not usual with solid textures, that are often globally defined by some procedural function.

One simple construction of a *space-to-color* function from samples has been introduced with the *reaction-diffusion* method of Turk [Tur91], who efficiently obtained a color evaluation at a given location using a simple weighted average of the neighboring samples, an idea later used in the Photon Mapping [Jen96]. Several commercial packages propose 3D brushes for texturing and modeling [Ali06, Rig06, Pix06] but do not address the problem of applying them on huge objects.

Another characteristic of our approach is to intensively use point-sampled geometry [PZvBG00, ABCO*01, PKKG03]. This permits to take benefit from most of the point-based tools for surface editing [ZPKG02, AWD*04].

2. Adaptive Out-of-core Simplification

The typical input of our algorithm is a polygonal soup [Lin00]. Alternatively, other large object representations can be used, like indexed meshes, by considering only the list of vertices, or point clouds (registered data sets from range scanner without surface reconstruction).

By sampling the original mesh, we are able to accurately select the resolution of the in-core object in order to target an interactive framerate, independently of the input model size. While some efficient out-of-core methods for resampling perform a mesh-to-mesh conversion, we rather think that, since this in-core object will be used only for intermediate processing, a more flexible representation, such as a *point set*, is a better choice. Furthermore, according to our texturing pipeline (see Figure 1), the in-core object will be used itself for defining a 3D texture. So, we do not want our final texture function depending of any underlaying surface topology produced by a given simplification method.

The last reason, but not least, is that the user must keep the control of the resolution, and be able to locally up-sample the in-core model from the original large one. In such a case,

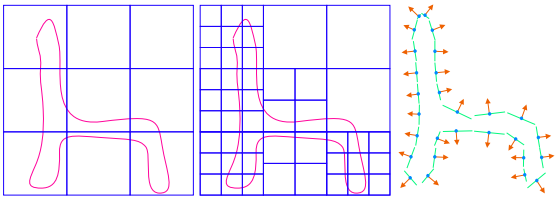


Figure 2: *Clustering by non-uniform m-grids. (a) The model is first partitioned in a coarse grid during an out-of-core streaming pass. (b) Then, a local sub-grid is generated in intersected cells, with a resolution chosen according to the local density. (c) Finally, a reduced set of samples is generated by clustering in a second out-of-core streaming pass.*

using a mesh-free representation, such as point sets, makes the insertion of new samples far more simple and efficient, while meshes would have required a complex local remeshing step, prone to artifacts.

To fulfil our constraints, we introduce a fast out-of-core simplification method. Ideally, this algorithm should handle two user-defined parameters: a *target size* tuned to ensure interactive framerate depending on the workstation capacities, and an *adaptivity factor* which controls the local density variation in the final simplified object, useful when the input large object is non-uniformly sampled.

We compose a new algorithm, somewhere between uniform clustering, such as the quadric-based simplification of Lindstrom [Lin00], and accurate adaptive ones, such as the octree-based clustering of Schaefer and Warren [SW03]. The former cannot easily ensure an output size in the case of non uniformly-sampled surfaces because of the fixed grid resolution, while the latter requires costly processes such as the pre-ordering of the large mesh and the intensive dynamic update of an in-core octree.

Out approach is based on the idea of *multi-grids*. A multi-grid of order m (noted *m-grid*) is defined as a tree of grids, where each node carries m^3 children, organized in a grid structure. For instance, a uniform grid of size 256^3 can be expressed either as a 256-grid of depth 1, a 16-grid of depth 2, a 4 grid of depth 4, or as 2-grid of depth 8 (the latter, being the classical octree). In fact, multi-grids can be decomposed in two families: *uniform multi-grids* [CP97] where the value m is the same for each node, and *non-uniform multi-grids*, where the number of children may vary between 0 and m for each node (see Figure 2). Basically, previous uniform clustering approaches can be seen as instances of a more general simplification scheme, based on *multi-grids*. We choose to use a two-pass algorithm to cluster our huge object into a *non-uniform multi-grid G* of depth 2.

First streaming pass: The first pass is performed in order to initialize the first level of G and to estimate, for each cluster i, the density γ_i (which is the number of input samples falling in the cluster i, see Figure 2(a)). As an input, the user provides the file where the mesh to texture is stored, a bounding box (i.e. the level 0 of the multi-grid) and a target size n for the in-core model. If the bounding box is not known, a preliminary out- of-core streaming pass is required. The resolution of the level 1 is m_0^3 (see Figure 2(a)), with $m_0 = \sqrt[3]{n}$. In the worst case of triangles randomly placed in the bounding box, this heuristic would lead to exactly n intersected clusters. However, our input objects are surfaces, which means that the number of intersected grid cells grows rather quadratically than cubically with the resolution. So, after this first filtering of the object through the memory, n_0 cells are intersected with $n_0 \ll n$ (in our tests, the following upper and lower bounds have been observed quite systematically: $n^{1/2} \leq n_0 \leq n^{2/3}$). We call these cells *1-nodes* (i.e. children of the multi-grid root). In order to speed up the remaining steps of the algorithm, all the input samples falling in the same 1-node are stored in a temporary file, attached to the node (see Figure 1).

Second streaming pass: Our sampling method for the second level has been inspired by quantification techniques in image processing (e.g. histogram equalization). For each 1-node i, we estimate a target sub-grid of size r_i (see Figure 2(a)), according to γ_i and a global user-defined value α. Actually, we map r_i on $[n/n_0 - \alpha, n/n_0 + \alpha]$ by setting:

$$r_i = \frac{n}{n_0} + \alpha\left(\frac{2\gamma_i - \gamma_{max} - \gamma_{min}}{\gamma_{max} - \gamma_{min}}\right)$$

The value α corresponds to the *adaptivy factor*: the larger α is, the higher the γ-variation of r_i will be. If $\alpha = 0$, there is no adaptivity and $r_i = n/n_0$ for all 1-nodes (uniform distribution assumption); in this case, our algorithm behaves like the Lindstrom one [Lin00], with the additional benefit that the high-resolution grid is generated only near the surface thanks to the first pass, which allows higher resolution for the same amount of memory. We instantiate a sub-grid of resolution $r_i^{3/2}$ (quadratic heuristic) for each 1-node. The cells of these sub-grids are called *2-nodes*. Then, we perform the second streaming pass from the temporary files, and cluster the samples according to the 2-nodes. At the end of the streaming process, each 2-node contains a final point-sample, storing its average position (either simply computed as the centroid of all the samples clustered in the node, or by using some quadric based approximation [GH97]), and optionally, its average normal, that will be used for in-core texturing (see Figure 2(c)). If there is no normal information in the input, it can be classically generated by a *principal component analysis* [HDD*92].

This efficient mesh-to-point-sample simplification algorithm reaches the target size specified by the user with less than 1% of error in all our tested examples, and our experimental results show that the *adaptivity* does not strongly influence this result.

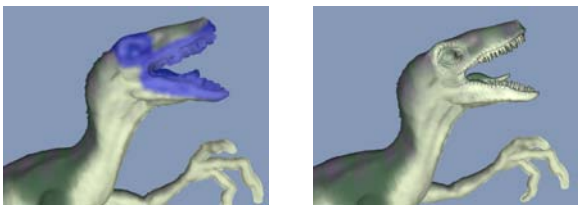


Figure 3: *Multi-scale painting.* ***Left:*** *After having roughly painted on it, the user selects an area (in blue) of the low-res sampled object.* ***Right:*** *A local refinement is performed, by up-sampling the selected area from the original large model. Newly inserted samples are textured according the current PST defined by the in-core point set; the user can now paint smaller features.*

3. Interactive multi-scale Texturing

The in-core point set can now be textured using flexible point-editing tools. We have chosen the PointShop3D software [ZPKG02] for defining the per-sample attributes interactively. Among other advantages, this system allows to apply bitmaps on the point cloud and to smooth-out features, as well as resampling the sample set according to the resolution of the applied textures. During the interactive texturing, the resolution of the in-core model can exhibit a lack of details for particularly accurate features. In this case, we re-use the temporary file generated at each 1-node during the sampling step (see Section 2) rather than streaming the whole original object, and up-sample the local area that requires more details. This illustrates the multi-scale behavior of our system: the user can initialize the texture at a global scale and then refine the model locally, while using the already defined points for inferring an initial PST for newly added points (see Figure 3). This construction uses the same point-sampled texture definition as for final surface coloring at full resolution, and it is described in the next section. The choice of point-based surfaces for intermediate representation is here very important: by avoiding any explicit topology, the local up-sampling does not require any local remeshing. Figure 3 gives an example of texturing with local up-sampling from the original large model for adding smaller details:

1. the user first selects the area to up-sample,
2. all 1-nodes of G are tested against this selection,
3. the sub-grids of intersected 1-nodes are refined,
4. the files associated to these nodes are streamed through these new sub-grids.

Alternatively, the user may specify to resample at the original resolution, and so **all** the samples of the files associated to the intersected 1-nodes are kept. The selection is removed and the set of samples obtained are inserted in the active point cloud and colored by the current PST, so far defined by in-core samples (see next section). The resolution of the in-core point cloud can also be increased in order to fit the resolution of the brush one [ZPKG02, AWD*04]. Figure 4 shows how a bitmap can be inserted in our PST at its **exact** resolution with this local up-sampling.

At any time, if the in-core model becomes itself too large for maintaining an interactive frame-rate, a down-sampling is performed, again on a per-1-node basis, by replacing the *Least Recently Used* (LRU) area by a unique sample, and storing the edited piece of surface on the disk. Latter, if the user comes back to this part of the object, the area is reloaded, and an LRU down-sampling is again performed until reaching interactivity. In pratice, it can be useful to maintain a ring of 1-nodes at current resolution around the currently edited piece of surface (i.e. 1-neighborhood safe, whatever the LRU selection). This simple LRU down-sampling rule makes the PST itself *scalable*.

4. Non-Uniform Point-Sampled Textures

Once the point set texture has been generated, the question is: "How to extrapolate the set of samples in order to use it at a higher definition?". Actually, this problem frequently arises in the field of *surface reconstruction*. In particular, *variational implicit surfaces* methods are ubiquitously recognized as quality approximation methods for a set of samples with attributes [TO02]. Usually, an iso-surface is finally extracted after fitting a function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ to the set of samples. In our case, the problem is simpler as we do not need iso-surface extraction, and just keep the function defining the implicit surface as a 3D texture.

Several function basis are available for filling the space with point-sampled attributes. *Radial Basis Functions* or *Moving Least Squares* [AGP*04] provide smooth 3D fields and can be evaluated locally. Unfortunately, in our case, the final evaluation of the function may potentially be done several hundred million times for either coloring the original file or directly shading pixels during ray tracing for instance. Thus, we rather adopt a simpler and more efficient approach that takes advantage of a very important feature of our PST: contrary to implicit surfaces used for geometric reconstruction, we do not need a signed value. In this case, a variation of the seminal idea of Turk for pattern creation [Tur91] can be adapted to our more general problem.

Figure 4: *Topology-free painting with up-sampling.* ***Left:*** *The integration of bitmaps in the low resolution model (100k samples) can be done either by coloring existing surfels (left rose) or locally up-sampling the model to reach the brush resolution (right rose).* ***Right:*** *The high resolution mesh (7M polygons) textured with the edited point set. Note the difference in sharpness between the two roses, when the up-sampling is performed.*

(a) (b) $k = 5$ (c) $k = 16$

Figure 5: *Point-sampled texture filtering. (a) A simple point-sampled texture. (b,c) Color texturing on a human face. The k-neighborhood used for space-filling intuitively drives the* smoothness *of the PST.*

We define a *point-sampled function* f_S over the point set S as follows: for each point p, $f_S(p)$ should return the corresponding texture attribute (usually color). Let $S_p = \{s_1, ..., s_k\}$ the k nearest samples of p. Each sample s_i handles a position p_i and an attribute c_i. We define the value $f_S(p)$ of the PST as:

$$f_S(p) = \frac{\sum_{i=1}^{k} \omega_p(p_i)\, c_i}{\sum_{i=1}^{k} \omega_p(p_i)}$$

Note that the size of the k-neighborhood influences the support radius of the reconstruction: a large value of k will smooth out the so-defined attribute function and can be used as an intuitive global *filtering* parameter for users (see Figure 5). In our implementation, k is user-defined. The function $\omega_p(p_i)$ is a *decay* function that controls the influence of the k-neighborhood of point p. A typical choice for ω is a Gaussian function, as in the *Confetti* system [PS04]. Nevertheless, in the context of large object texturing, selecting a less computationally intensive function is often interesting. We choose the standard uniform cubic hermite polynomial, usually recognized as a good and fast approximation of Gaussian-based kernels:

$$\forall t \in [0,1] \quad h(t) = 1 - 3t^2 + 2t^3$$

The kernel function $\omega_p(p_i)$ uses the previous polynomial simply adapted to S_p, and is hence defined as:

$$\omega_p(p_i) := h\left(\frac{|p - p_i|}{\max_k(|p - p_k|)}\right)$$

Note that the kernel function is extremely inexpensive, but the *feature preserving* control by k may filter out some high frequency details present in the texture. When this is an issue, *singular weight kernels* (i.e., Dirac behaviour near zero) can be used. Alternative feature-preserving kernels may also be chosen among the huge set of kernels developped over the years, in the image processing community.

Texture Antialiasing: The up-scaling of the PST does not exhibit artifacts thanks to the smooth filtering provided by the kernel function. However, in the case of ray-tracing, when the texture is directly used for evaluating the color of a pixel, the down-scaling of the PST may lead to aliasing. Using cone tracing instead of ray tracing is a common (but expensive) solution to prevent such aliasing. In our case, cone intersection can be speeded up by replacing the evaluation of $f_S(p)$ by the average of the samples falling in the sphere Φ, centered at the intersection point. The diameter of Φ is chosen as the object-space size of the pixel at the intersection point. This special evaluation is performed as soon as more than one ray sample intersects Φ.

5. Implementation and Results

We have implemented our framework as a plug-in for PointShop 3D. In all our tests, we use a PiV Intel 2.4GHz with 512 MB of memory and an UDMA hard-drive.

Implementation: When PST look-up is mandatory (e.g. refinement from temporary files or final texturing of the original large model), a kD-Tree is built over the current in-core point-set. This structure allows a fast k-neighborhood query for finding S_p. The typical size of our in-core point set is between 100k (after out-of-core simplification) and 500k (worst case observed after all the local refinements involved in a whole texturing session). This induces a very fast generation of such a tree (less than one second in all our tests). Our *m-grid* resampling scheme is implemented on a pointer-based tree, where each node carries a reference to its children nodes, as well as an average texture value. Note that only the 1-nodes (i.e. root and its children) are stored in memory during texturing (local sub-grids are instanced only at refinement time). Most steps of the algorithm (second streaming pass during sampling, interactive refinement during texturing, and feedback of the texture on the initial model) are dealing with the set of temporary files generated at loading time (see Figure 1) and are implemented with *multi-threads*. This allows to take benefit from multi-core CPUs and multi-CPUs that are more and more present on common workstations.

Performance and Analysis: The feedback on the high-resolution object is really important when texturing its low-resolution version. So, a key property of an out-of-core texturing method is to be able to efficiently apply the generated texture on the initial object, in order to offer the user a fast *quality control* of his work. Table 1(a) gives the sampling time for different models. Globally, our simple sampling scheme is limited by the hard-drive when reading the data (about 60% of the total processing time), and not by any update of data structure, since nodes are created statically, and never removed or collapsed during the interactive process [SW03].

While our simple density-based filtering does not bound curvature error in the simplification, we still obtain better results than the Lindstrom algorithm [Lin00], for a computational cost which is dramatically reduced compared to the arbitrary depth octree technique of Schaefer and Warren [SW03] (see Section 5). Note that a bounded curvature

error is possible if wished by the user, by simply employing a third out-of-core streaming pass. However, since we dot not perform an *extreme* simplification, this is not necessary.

Globally, for the same number of final samples, our experimental results on the different models used in this paper show that our approach is about two times slower than the grid method of Lindstrom [Lin00] but provides much nicer results. Note that the Lindstrom method is not easily amenable to local interactive refinements, intensively used in our system. Similarly, our approach is about five times faster than the octree-based method of Schaefer and Warren [SW03].

Models	Tri.	Time
Lion	6.5*M*	3.85s
Raptor	8*M*	4.69s
David	56*M*	16.5s
StMatt.	372*M*	190.5s
Atlas	500*M*	276.9s

(a)

Samples	Evaluations	
	4M	**28M**
4504	2.0*s*	15.1s
15216	3.7*s*	21.7s
49482	4.9*s*	29.1s
146686	5.7*s*	35.5s

(b)

Table 1: *(a) Timings for out-of-core simplification. The target size was set to 100k point samples, and was reached in all cases with less than 1% of error. (b) Texture look-up timings for a given number of evaluations over a sampling at a given resolution.*

Table 1(b) exhibits the look-up time with a target large model at two different resolutions. Obviously, the average look-up time is independent of the input model size. But more surprisingly, it appears that the size of the internal point-set does not strongly influence the average look-up time. Actually, in practice, the kD-Tree query remains a low-cost operation for the size of our typical in-core point sets. This can be explained by the fact that large models already contain fine features, more particularly in the normal field. Thus, in practice, users will not have to "trick" the texture for obtaining a more complex visual effect when painting, and will only focus on surface color at a different resolution as mentioned by DeBry et al. [gDGPR02]. In other words, a large part of what we usually call "visual detail" is already present in the huge geometry of input objects (see Figure 8).

Comparison: To our knowledge, no system has been proposed for interactive painting on huge objects that do not fit into memory. Nevertheless, among the contributions of this paper, PST can easily be compared to octree textures. Basically, the main advantage of PST over octree textures was to allow the user to interactively refine directly from the original surface, without being constrained to the grid topology induced by octrees (See Figure 4). Simple point sets allow greater flexibility and very quick variation in the density of sampling (which are very frequent when the user wants to texture a given area more accurately [gDGPR02]) where a very deep octree would have been necessary. Last but not least, octree textures cannot represent efficiently fine color features which are not axis-aligned. However, the uniform structure of octree textures allows efficient on-GPU implementations [LHN05], which is more difficult for our non-uniform point-sampled textures. Of course, in such a situation, our PST can be straightforwardly resampled in an octree texture for real-time shading. But, we rather focus on very large objects, for which the color is usually encoded in the data-structure, on a per-sample basis, for efficient rendering [RL00, DVS03, GM05]. For instance, objects shown on Figure 6 and 7, rendered with QSplat, have been generated from large polygonal meshes and textured on a per-vertex basis.

6. Conclusion and Current Work

We have proposed a new simple method for interactive texturing of large objects. Our method produces a 3D texture function, interactively defined over a set of samples of the original large object. By introducing *point-sampled textures* as flexible texture definitions, our framework proposes a novel simple and multi-scale texturing system, and can be either used for fast rough painting or precise interactive texturing of large objects. Produced textures are *output-sensitive* and do not depend of the input large object size. All existing and future point-based texturing tools can be used for painting the models. Convincing results have been obtained either for coloring large meshes in the context of real-time visualization (see Figure 6 and 7) or defining more complex shading for high-end computer-graphics (Figure 8).

The major drawback of our system is also its strength: this is a *parameterization free* tool for texturing large objects, which means flexibility and efficiency as demonstrated throughout this paper, but which also implies that its practice is slightly different from usual 2D painting softwares [Ado06] and requires for artists to change their habits. This is also the reason why 3D painting is still an active research field: retrieving in 3D the accuracy of popular 2D painting packages **is** a challenge that would also induce new interaction metaphors. Future work includes the extension to interactive multi-scale freeform modeling of gigantic objects, still using points as an intermediate representation. We also plane to investigate appearance multiresolution in the PST itself as well as *on-the-fly* spectral analysis of the PST.

Acknowledgments We thank the Computer Graphics Group of ETH Zurich for providing PointShop 3D, and the Digital Michelangelo Project and the Aim@Shape network for providing models (Aim@Shape models are modified).

References

[ABCO*01] ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C. T.: Point set surfaces. *IEEE Visualization* (2001).

[Ado06] ADOBE: Photoshop, 2006.

[AGP*04] ALEXA M., GROSS M., PAULY M., PFISTER H., STAMMINGER M., ZWICKER M.: Point-based computer graphics. *ACM SIGGRAPH Course* (2004).

[Ali06] ALIASWAVEFRONT: Maya, 2006.

[AWD*04] ADAMS B., WICKE M., DUTRÉ P., GROSS M., PAULY M., TESCHNER M.: Interactive 3d painting on point-sampled objects. In *Point-Based Graphics* (2004).

[BD02] BENSON D., DAVIS J.: Octree textures. In *ACM SIGGRAPH* (2002).

[CD05] COOK R. L., DEROSE T.: Wavelet noise. In *ACM SIGGRAPH* (2005).

[CP97] CAZALS F., PUECH C.: Bucket-like space partitioning data structures with applications to ray-tracing. In *ACM Symposium on Computational Geometry* (1997).

[DVS03] DACHSBACHER C., VOGELGSANG C., STAMMINGER M.: Sequential point trees. *ACM SIGGRAPH* (2003).

[gDGPR02] (GRUE) DEBRY D., GIBBS J., PETTY D. D., ROBINS N.: Painting and rendering textures on unparameterized models. In *ACM SIGGRAPH* (2002).

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *ACM Siggraph* (1997).

[GM05] GOBBETTI E., MARTON F.: Far voxels. *ACM SIGGRAPH* (2005).

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surface reconstruction from unorganized points. In *ACM SIGGRAPH* (1992).

[HH90] HANRAHAN P., HAEBERLI P.: Direct wysiwyg painting and texturing on 3d shapes. In *ACM SIGGRAPH* (1990).

[Jen96] JENSEN H. W.: Global illumination using photon maps. *Rendering Techniques* (1996).

[LHN05] LEFEBVRE S., HORNUS S., NEYRET F.: *GPU Gem's 2: Octree Textures on the GPU*. 2005.

[Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *ACM SIGGRAPH* (2000).

[Per85] PERLIN K.: An image synthesizer. In *ACM SIGGRAPH* (1985).

[Pix06] PIXOLOGIC: Z brush, 2006.

[PKKG03] PAULY M., KEISER R., KOBBELT L. P., GROSS M.: Shape modeling with point-sampled geometry. *ACM SIGGRAPH* (2003).

[PS04] PAJAROLA R., SAINZ M.: Confetti: Object-space point blending and splatting. *IEEE TVCG* (2004).

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: Surface elements as rendering primitives. In *ACM SIGGRAPH* (2000).

[Rig06] RIGHTHEMISPHERE: Deep paint 3d, 2006.

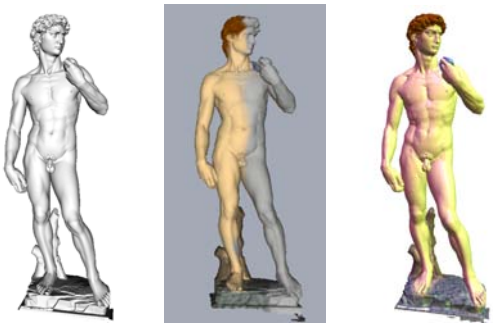
[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: a multiresolution point rendering system for large meshes. *ACM SIGGRAPH* (2000).

[SW03] SCHAEFER S., WARREN J.: Adaptive vertex clustering using octrees. In *Proceedings of SIAM Geometric Design and Computing* (2003).

[TO02] TURK G., O'BRIEN J. F.: Implicit surfaces that interpolate. In *Proceedings of Shape Modeling International* (2002).

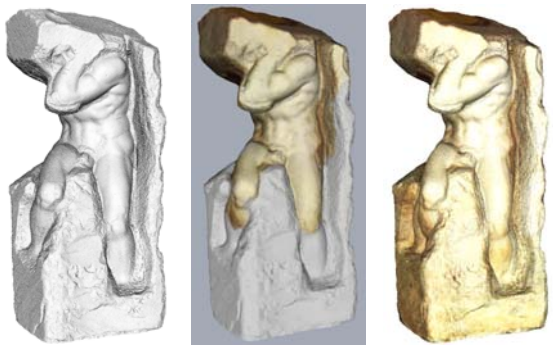
[Tur91] TURK G.: Generating textures on arbitrary surfaces using reaction-diffusion. In *ACM SIGGRAPH* (1991).

[ZPKG02] ZWICKER M., PAULY M., KNOLL O., GROSS M.: Pointshop 3d: An interactive system for point-based surface editing. In *ACM SIGGRAPH* (2002).



(a) David (56M polygons) (b) 200k samples for interactive texturing (c) 56M textured polygons

Figure 6: *Interactive multi-scale texturing of the David model. (a) Original large mesh. (b) Interactive multi-scale texturing with our system. (c) Application of the PST to the original model and real-time visualization (QSplat).*



(a) Atlas 500M polygons (b) 275k samples for interactive texturing (c) 500M textured polygons

Figure 7: *Recoloring Michelangelo's Atlas. (a) Original uncolored large mesh. (b) Interactive multi-scale texturing with our system, using several photos from the original statues and a sample set of stone textures. (c) Application of the PST to the original large model and real-time out-of-core visualization.*

Figure 8: *Point-sampled texture for high-quality rendering. Top left: original mesh (6.5M polygons). Bottom left: diffuse and specular interactive multi-scale texturing with our system (50k point samples). Right: offline rendering of the original mesh (6.5M textured polygons) with our point-sampled textures (diffuse and specular component).*