The 13th International Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST (2012)
D. Arnold, J. Kaminski, F. Niccolucci, and A. Stork (Editors)

Interactive exploration of gigantic point clouds on mobile devices

Marcos Balsa Rodriguez, Enrico Gobbetti, Fabio Marton, Ruggero Pintus, Giovanni Pintore, Alex Tinti

CRS4 Visual Computing, Italy – http://www.crs4.it/vic/

Abstract

New embedded CPUs that sport powerful graphics chipsets have the potential to make complex 3D applications feasible on mobile devices. In this paper, we present a scalable architecture and its implementation for mobile exploration of large point clouds, which are nowadays ubiquitous in the cultural heritage domain thanks to the increased performance and availability of 3D scanning techniques. The quality and performance of our approach is demonstrated on gigantic point clouds, interactively explored on Apple iPad and iPhone devices using in variety of network settings. Applications of the technology include on-site exploration during scanning campaigns and promotion of cultural heritage artifacts.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.2]: Distributed/network graphics—Computer Graphics [I.3.7]: Three-dimensional graphics and realism—

1. Introduction

The rapid evolution of shape and color acquisition technologies, from active scanning to computational photography methods, is making large amounts of high-quality sampled 3D data available, especially in the field of cultural heritage (CH) where artifacts are routinely scanned for preservation, study, or presentation.

Detailed 3D models are non-trivial to render since they require fast incremental loading, reasonable compression, dedicated GPU rendering approaches, and adaptive view-dependent culling techniques. For these reasons, interactive rendering of these huge datasets remains a very challenging problem. The visualization of such massive models is typically addressed using level-of-detail (LOD) and asynchronous out-of-core data fetching coupled with parallel rendering techniques, while the most commonly used data representation for CH data sets is the triangulated mesh. However, recently the interest in the use of point clouds has grown, since these are easy to build and manage. Points as rendering primitives are more efficient and result in a more compact representation, since the mesh connectivity of triangles is not required. More benefits arise thanks to the simplicity of pre-processing algorithms, and the reduced amount of data transferred in client/server applications for remote 3D visualization. Thus, direct point-based rendering has gradually emerged as a useful tool to interactively inspect very large geometric models [GP07]. At the same time, the rapid evolution of low-cost graphics hardware has made 3D model visualization available on very different platforms, from laptop PCs to mobile devices. However, while a lot of solutions have been presented for desktop and laptop platforms [GKY08], distributing and rendering non-trivial 3D models on portable device is still challenging, since it is subject to strong limitations in terms of 3D hardware capabilities, memory, storage and network bandwidth.

Our contribution is a point-based rendering solution for remote high-quality interactive visualization of massive static point clouds on consumer-grade portable devices. The presented client-server framework gives the user the possibility to navigate and inspect the full-resolution model in real-time, using a limited bandwidth. Our technique is easy to implement, provides good performance and quality rendering of massive point clouds, and performs an efficient massive data distribution with a high level of scalability. Although not all the techniques presented here are themselves novel, their combination in a single system for remote rendering on mobile platforms is non-trivial and represents a substantial enhancement to the state-of-the-art.

2. Related Work

In this section we briefly discuss the approaches that are most closely related with this work. Readers may refer to

well-established surveys [KB04, GKY08, CPAM08] for further details.

While many examples exist for rendering light 3D models on portable platforms (e.g., MeshPad [IST12] for meshes or PCL [Mar12] for points), exploring massive models on mobile devices is still a hot research topic. However, much of the work in model distribution has focused so far on compression rather than adaptive view-dependent streaming. MPEG-4 is a reference work in the field [JPP08]. Early methods for view-dependent LOD and progressive streaming of arbitrary meshes use fine-grained updates based on edge collapses or vertex clustering [XV96,Hop97,LE97]. Many compression and streaming formats for the web have been built with these approaches [MLL*10,BCK*11,NKB10], but they are CPU-bound and spend a great deal of rendering time computing a view-dependent triangulation prior to rendering, making their implementation in a mobile setting particularly challenging. Recently, Gobbetti et al. [GMB*12] proposed an efficient image-based mesh representation; however, it only works for models for which an isometric quad parametrization exists. In contrast, we focus here on a more practical multi-resolution representation for point clouds.

While the use of points as rendering primitives was introduced very early [LW85], only over the last decade have they reached the importance of fully established geometry and graphics primitives [KB04]. Since its inception, many techniques have been presented for improving the display quality, LOD rendering, as well as for efficient out-of-core rendering of large point models. Streaming QSplat [RL01] has for a long time been the reference system for massive point count rendering in a network setting. However, the algorithm is CPU-bound since all the computations are made per point, and CPU-GPU communication requires a direct rendering interface. While reasonable performance has been demonstrated on early PDAs [DD04,HL09], the methods are hardly applicable to current devices with high-definition screens, due to the high per-point costs. A number of authors have thus proposed to raise performance limits through coarse-grained structures and efficient usage of retained-mode rendering interfaces.

Layered point clouds (LPC) [GM04a, GM04b] and Wand et al.'s out-of-core renderer [WBB*08] are prominent examples of high-performance GPU rendering systems based on hierarchical model decompositions into large-sized blocks maintained out-of-core. LPC is based on adaptive BSP subdivision, and subsamples the point distribution at each level. In order to refine an LOD, it adds points from the next level at run-time, limiting its applicability to uniformly sampled models and producing moderate quality simplification at coarse LODs. In Bettio et al.'s approach [BGM*09] these limitations are removed by making all BSP nodes self-contained and using an iterative edge collapse simplification to produce node representations. In this work we propose an improved BSP construction method which uses a split plane position optimization, as well as a rendering architecture tuned for mobile devices. In contrast, Wand et al.'s approach [WBB*08] is based on an out-of-core octree of grids and deals primarily with grid-based hierarchy generation and editing of the point cloud. Its limitation is in the rendering quality of lower resolutions created by the grid, no matter how fine it is. Goswami et al. [GEM*12] recently proposed a technique based on multi-way kd-trees which, by controlling a node's fan-out, simplifies memory management by creating uniformly sized nodes managed out-of-core in a memory-mapped array. Unfortunately the method looses much of its appeal in a networked setting, where variable bitrate compression is typically used at the node level. None of the previous approaches have thus far been implemented on mobile devices.

3. Method overview

Given an input model, we build a multi-resolution structure based on its binary space partition. The structure has approximately the same number of samples (few thousands) in the leafs and in the inner nodes, which are constructed by filtering the two child samples. This structure is pre-processed off-line, starting from a point cloud model. The data is recursively partitioned up to the leafs using an out-of-core approach. Inner levels are built merging children with a simplification strategy that aims to have a target number of equally spaced samples per node. Multiple clients can access a server farm on which the models are stored. Each client adapts the representation of the model depending on the viewing parameters, incrementally updating the working set used for rendering. The algorithm tries to keep the working set occupancy constant with a LRU strategy, never going beyond the allocated memory resources. Data is accessed through a data access layer which takes care of the communication with the server. Rendering is simply a matter of drawing all the vertex buffer objects associated with the view.

4. Pre-processing

The pre-processing phase consists of two main steps: the first one partitions the input dataset as a kd-tree, while the second produces the node representations.

In the first step, the input point cloud is inserted into an out-of-core array, which is recursively split down to the leaves. Nodes are split if they contain more than the target number of samples. The split plane has to be selected appropriately, taking into account conflicting constraints. First of all, nodes should tightly bind geometries, excluding empty space. Moreover, since at run-time a constant resolution will be employed for rendering each node, they should be as compact as possible, with approximately the same size over the three dimensions. Finally, to reduce tree depth, points should be uniformly split among a node's children. Unlike Goswami et al. [GEM*12], who employ multi-way kd-trees

but fixed plane positions, here we optimize the split plane position to balance the various constraints. To quickly achieve this goal, we scan the point clouds generating split planes that produce acceptable aspect ratios, and from these we choose the plane that produces the most balanced partition.

During the second step, inner nodes are constructed bottom up starting from the leafs. Each node is built by merging the samples contained in two children into a single sample set, before applying a simplification process to reduce a node's point could to the target sample count. The simplification procedure uses a local kd-tree containing all the samples of the two children, and applies a multiple choice decimation technique [WK02]. At each decimation step, we randomly select a small number (8) of points in the kd-tree and look for the closest neighbor with a compatible normal. We then select the pair with the shortest distance, merging it into a single point. The procedure is repeated until the point cloud reaches the target point count. The color and normal attributes of the collapsed samples are interpolated with a weight proportional to the sample splat area, while the radius is the minimum radius that covers both original samples from the new sample position. The decimated point cloud is then compressed to reduce run-time bandwidth usage and the node is stored in an out-of core database.

Instead of looking for maximum compression, we employ a compression algorithm which is fast enough to be decompressed on a low-powered device. Each node contains an header and a point cloud. In the header we store a flag indicating if it is a leaf, the bounding box, the radius range (min,average,max) and the total point count. The point cloud is stored in four parallel arrays: positions, normals, colors and radii. Compression is based on a simple wavelet compression scheme which is characterized by fast decompression and reasonable compression rates. Before applying the wavelet transformation the points are sorted according to an order that minimize the Euclidean distance among adjacent points. The attributes of the points in the point strip are then stored in the rows of a 2D array. These rows are then transformed using a reversible n-bit to n-bit wavelet based on the Haar wavelet transform in order to reduce entropy [SLDJ04]. The low-pass coefficients produced by the transformation are iteratively filtered by this wavelet step until we remain with the root main single coefficient. We store the root and all the detail coefficients at various levels. The resulting coefficients are then mapped to positive integers and encoded using a simple Elias gamma code [Eli75], in which a positive integer x is represented by: $1 + \lfloor \log_2 x \rfloor$ in unary (that is, $\lfloor \log_2 x \rfloor$ 0-bits followed by a 1-bit), followed by the binary representation of x without its most significant bit. Upon decompression, all the steps are undone in the reverse order, with the obvious exception of sorting. To transform the input data to coefficients, an adaptive quantization step that considers the local sample spacing is employed: the positions are expressed relative to the node bounding box and are quantized to the minimum number of bits per component that is able to produce a quantization error under a quarter of the minimum sample radius. The same error threshold is used to quantize radii expressed inside the radius range stored in the header. To eliminate any holes the radii are subsequently enlarged by the maximum quantization error. We exploit a radial projection to compress normals to 16 bits, while colors are first remapped to YCoCg-R color space [MS03] to reduce correlation and then mapped to 5 bits for the luminance and 6 bits for each chroma components. Since the compression subsystem is used by both the pre-processing and the run-time systems, quantization effects in a given node are taken into account when constructing its parent.

5. Server

The server stores different databases, each one corresponding to one model. In order to increase server-side scalability, no processing component is present in the server. This approach makes it possible to leverage existing database components instead of being forced to implement a storage manager. For this work, we use Berkeley DB for storing, accessing and caching data, while we developed a custom module for Apache2 to serve data. We employ Berkeley DB because it is an embeddable database that is open source and widely deployed, and because of its technical characteristics; namely, it provides a fast, scalable, transactional database engine, and it is able to manage up to terabytes of data. Moreover, the amount of per-process replicated cache is configurable, and different instances of the same database are able to share index memory, thus reducing memory load for servers while handling multiple clients in parallel. Likewise, we chose to serve data with Apache2 since it is an efficient, extensible and secure open source web server which provides many HTTP services adhering with the current HTTP standards. In addition, it is scalable, multi-threaded and includes features like persistent server processes and proxy load balancing, which are essential for the scalability of our application. A custom Apache2 module manages our application's connectionless protocol, based on HTTP. Client requests are simply made by the database name and node identifier. The module parses the request, fetches the associated data from the DB, and if data is present, returns the compressed node data, otherwise it returns an empty message. The proposed approach relies on widely tested components, is simple to implement and provides very good performance, particularly thanks to the concurrency features of Apache2 which it leverages to manage up to thousands of clients in parallel by forking multiple child servers sharing the same database.

6. Client architecture

Modern mobile devices such as the Apple iPhone, iPad and other devices on other platforms, such as Android, typically offer support for OpenGL ES, the OpenGL specification for embedded systems. The versions of OpenGL ES

that are supported in the current generation of mobile devices are 1.1 and 2.0. With the ES 1.1, which is based on OpenGL 1.5, VBOs and vertex arrays are the only methods to submit geometry primitives (polygons are not available as primitives). Complex texturing (multiple textures and mipmapping), point sprites and clipping planes are supported. Some devices also offer support for Frame Buffer Objects and non-power-of-two textures as an extension. On the other hand, in the ES 2.0, which is based upon OpenGL 2.0, all the fixed pipeline functionality has been removed and vertex and fragment shaders must be provided, which gives more flexibility. More control on data precision has also been added to the shading language. Frame buffer objects are included in the specification while user clip planes have been removed together with the fixed functionality.

The GPUs that are typically implemented in these devices are focused on high efficiency and low power consumption. For instance, the PowerVR SGX5XX used in the various iPod, iPhone and iPad series, offers a fully programmable hardware pipeline typically using tile-based deferred rendering (TBDR). Only once all the primitives have been submitted the driver splits the geometry into tiles that are then rendered using a small amount of in-core memory. Due to this architecture, reading back from the frame buffer is a costly operation since it requires all the tiles to be written to the frame buffer prior to reading from it. Most of the GPUs used in mobile devices use this type of rendering technique. In general, the GPUs included in current mobile devices are very efficient and give really good performance; however, since the display resolution is constantly increasing, the fragment load heavily penalizes the rendering (e.g., current iPad 3 devices are already offering resolutions over HD with 2,048 by 1,536 pixels).

For these architectural reasons, we have thus designed the run-time rendering engine to work using direct rendering, communicating data in large batches using cached vertex arrays, and reduced the usage of fragment shaders. We have also decoupled rendering from adaptive refinement and data fetching using a multi-threaded approach.

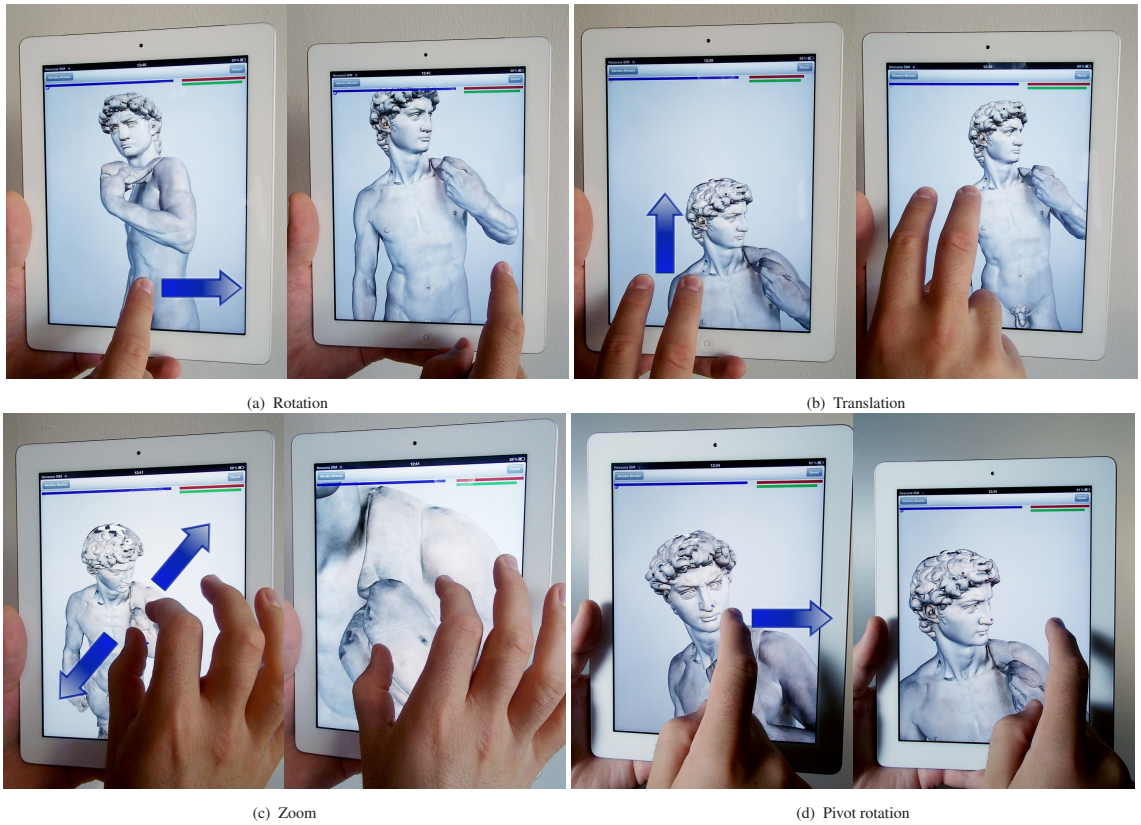
6.1. Adaptive view-dependent representation

Given viewing parameters and a fixed screen space tolerance, the client performs adaptive rendering of the multiresolution model, which is incrementally fetched from a server. The view-dependent representation of the hierarchical structure is constructed coarsely and then refined. The traversal algorithm of the kd-tree takes into account several factors: the user's point-of-view, the available GPU resources on the mobile device platform, the current CPU usage level, and the network bandwidth required for streaming the data from the remote repository to the mobile device. The model is represented by a coarse-grained kd-tree structure, where for each node there is a pointer to a data object – a decompressed vertex array inside a LRU cache. The cache of vertex arrays is directly used for rendering, but it also keeps a number of unused decompressed nodes, which could be useful to refine previously visited branches of the hierarchy. We hide network latency by using asynchronous data requests. The refinement technique exploits the progressive nature and coarse granularity of the multi-resolution hierarchy to reduce CPU processing costs. At each frame, this structure is adapted to the current point of view by incrementally updating the current working set that approximates the model.

The refinement process is driven by a user-defined pixel threshold, which represents the required average sample distance between adjacent splats on the screen. The algorithm performs a single-pass recursive traversal of the multiresolution structure and selects the nodes that need to be included in the working set. For each node, we test whether the bounding box of the node is completely outside the frustum view. If the node bounding box is outside, the traversal stops discarding the entire branch of the tree. Otherwise, we continue the recursive refinement of its children. The node average sample distance is projected into the screen to obtain the average splat size. A reasonable upper bound for the projected size is obtained by measuring the projected size of a sphere with diameter equal to the average sample distance in object space and centered at the bounding box corner closest to the viewpoint. For each node, we compare the projected average splat size with the threshold; if the value is under the threshold, the corresponding point cloud within the node is prepared for rendering and the sub-tree underlying the node is coarsened to remove overrefined data. Otherwise, we try to refine the node. In order to avoid freezing the renderer because of data access latency, we first check the availability of the children nodes. If children are available, we proceed with the refinement. Otherwise, we select the node for rendering. Once the refinement is complete all the selected nodes can be rendered.

6.2. Multi-threaded data access layer

The adaptive loader retrieves data through an asynchronous data access layer, which encapsulates the data fetching mechanism and does not block the application when data is not immediately available. The data access component runs in two threads that communicate through a shared cache of compressed nodes, which are identified by their node ids. Nodes are kept in compressed form to increase the node capacity of the cache and thus reduce cache misses and better shield the application from network latency. The main application thread asks for the data required to complete the refinement of a node. If the data is available, the node is returned and the refinement continues; otherwise, a request for the data is pushed into a priority queue and the refinement process for that node is paused. The requests pushed into the queue are served by the second thread, which takes care of sending a certain number of requests (as many as can be reasonably



(a) Rotation (b) Translation

(c) Zoom (d) Pivot rotation

Figure 1: ***Graphical User Interface.*** *Here we show the multi-touch gestures to navigate massive 3D models on iPad and iPhone platforms: (a) rotation of the model about its bounding box; (b) translation; (c) zooming; (d) rotation of the model around a target (blue sphere) activated by a single quick tap.*

satisfied by the estimated available bandwidth). The remaining requests are ignored. Since the requests are ordered by their priority, which corresponds to the node's projected error, coarser nodes are served first. Moreover, discarded requests that are still needed will be posted on the next frame. A simple limited memory first-in/first-out queue results in a request ordering that is I/O-efficient and ensures that the most relevant data is downloaded as soon as possible. Finally, the second thread also manages the incoming compressed bit-streams and storing them in the shared cache. In our current implementation, we use a HTTP/1.1 persistent connection and optionally employ HTTP pipelining. The combination of these two techniques improves bandwidth usage and reduces network latency. It also allows us to keep the API for the protocol simple, since the client benefits from a connection-based implementation hidden behind a reliable connectionless interface. The use of HTTP pipelining allows multiple HTTP requests to be sent together, without waiting for the corresponding responses. The client then waits for the responses to arrive in the same order in which the corresponding requests were sent. Request pipelining can result in a dramatic improvement in response times, especially over high-latency connections.

6.3. Rendering process

Once the incremental refinement is finished, the nodes in the working set are the view-dependent 3D model approximation for the current frame. Since there are a variety of possible clients with different graphics capabilities, we have to detect the underlying available GPU resources at the beginning of the rendering stage. Then, we set graphics parameters of our algorithms accordingly. Communication with the GPU is performed exclusively through a retained mode interface by managing a cache of vertex arrays in the GPU. It exploits spatial and temporal coherence by reusing the same data for several frames without the need to reload it again. Our architecture supports two rendering modes. The first is a simple approach which renders circular splats through *glPointSmooth* and uses the same splat size for all the samples in the point cloud. The second is a higher quality representation, implemented using vertex and fragment shaders, that draws an oriented 3D circle depicting a textured quad for

Figure 2: ***Precomputed view positions.*** *The interface provides the possibility of browsing a list of pre-computed view positions (bottom side of the interface).*

each sample [BK03]. The latter solution tries to simulate a smooth surface by drawing each splat with its own size and orientation. The splat has a size proportional to its average sample distance, and it is orthogonal to the sample normal.

6.4. Graphical user interface

The Graphical User Interface (GUI) is based on the Cocoa Touch UI framework and it is composed of a Model List Widget and OpenGL Rendering Widget. Through the Model List Widget the user can browse and select the desired model. Then, the OpenGL rendering widget allows the user to explore the selected model through standard multi-touch gestures. Specifically, the user can rotate the model about its bounding box, center it by moving a single finger on the screen (Fig. 1(a)), translate it by moving two fingers on the screen (Fig. 1(b)), and finally zoom in and out by performing a pinch gesture (Fig. 1(c)).

An alternative "target-based" navigation approach is also provided. With a single quick tap the user can select a target point, which will be shown as a small sphere attached to the model. The user can then rotate the model about the target sphere (Fig. 1(d)). Further, a quick tap on the sphere will automatically animate the camera from the current position toward target position. The target can be deactivated by tapping outside the small sphere. For camera motion, the multi-touch interaction system has been enriched with the possibility to browse a list of pre-computed view positions (Fig. 2), displayed as a series of thumbnails in the lower part of the screen. When a view is selected, the camera is smoothly animated from the current position the newly selected view.

7. Results

The method presented was used to develop a C++ application composed of a pre-processor, a client application and a server. Several tests were performed on pre-processing and rendering of very large models (see Fig. 3 and 4). In Table 1 we list the 3D models used for testing; these datasets are the last version of the high resolution scan of the statues of David (about 500M samples) and St. Matthew (about 200M samples) from the Digital Michelangelo Repository of Stanford University. The first model has a color signal acquired after a restoration process. The color was blended on the geometry using the algorithm by Pintus et al. [PGC11]. The St. Matthew data has an ambient occlusion gray scale attribute per point. They are all acquired using triangulation laser scanning with a sub-millimeter accuracy. Using these models, we first built an out-of-core multi-resolution hierarchy, which was stored on the server side and was fetched by a remote client application. This pre-processing was performed using an off-the-shelf PC with Linux 3.0.6 (Gentoo distribution) and an Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz with 24GB RAM. The David model was processed in about 240 minutes and generated a multi-resolution model made of 470M samples. The St. Matthew model was processed in about 80 minutes and generated a multi-resolution model of 190M samples. These run times are comparable with state-of-the-art pre-processing methods for massive point cloud rendering [BGM*09, WS06, WBB*08].

Model	Resolution	Pre-processing time
St. Matthew	190Mpoints	80mins
David	470Mpoints	4hours

Table 1: ***Datasets and pre-processing.*** *We tested our algorithm on two CH datasets: the David and St. Matthew statues from the Digital Michelangelo project. Here we list the size in points and the time spent to pre-process the data to obtain the multi-resolution hierarchy stored on the server side.*

The client was implemented on iOS 5 using C++, OpenGL and Objective-C++ and the rendering tests were performed on an iPhone 4 and on "the new iPad". The iPhone has a 1Ghz Apple A4 processor with 512 MB RAM, a PowerVR PowerVR SGX535 GPU and a screen resolution of 640 x 960 pixels, while the iPad has a 1Ghz Dual-core Apple A5X processor with 1GB RAM, a PowerVR SGX543MP4 GPU and a screen resolution of 2048 x 1536 pixels. Rendering was tested in a variety of scenes ranging from far view to near close up, including abrupt rotation to test the capabilities of the algorithm. For both iPhone and iPad, we can sustain an average rendering frame rates above 30 fps, never going below the interactive rate of 15 fps. The iPad hardware is more powerful than the iPhone, but it has to render much more points due to its high screen resolution. For this reason, the perceived performance on these two devices is very similar. Multi-resolution and view frustum culling are able to discard unneeded data, keeping a working set of 1M points. The data compression rate is about 4.2bytes/sample for the colored David model and 3.6bytes/sample for the St. Matthew, since it only has a single gray scale value for the ambient occlusion vertex attribute.

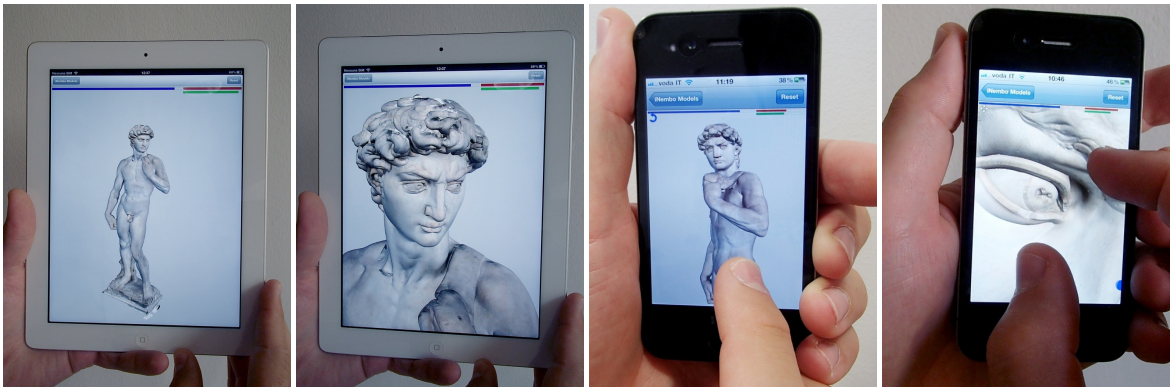


Figure 3: ***Colored David 0.25mm dataset.*** *A colored 470M-point 3D model rendered with iPad and iPhone devices. The model is from the Digital Michelangelo project, courtesy of Marc Levoy and the Soprintendenza ai beni artistici e storici per le province di Firenze, Pistoia, e Prato.*

Connection	peak (MB/s)	average MB/s)	iPad full refinement (David/St.Matthew)	iPhone full refinement iPhone start-up (David/St.Matthew)
ADSL 8Mpbs	6.4	3.2	1.5sec / 2sec	<1sec
UMTS/HSPA	1.7	1.3	3.5sec / 6sec	<1sec
EDGE	0.023	0.02	225sec / 375sec	12sec / 22sec

Table 2: ***Client network performance.*** *Given three types of wireless network, we present their peak and average bandwidths and the start-up full refinement time for the David 470Mp and St. Matthew 190Mp datasets using both Apple iPad 3 and iPhone 4 platforms.*

One of the most critical issue in mobile applications is the time needed to fully refine the data at application start-up and to refine the model during the navigation. This time is independent from the rendering thread and depends on the network bandwidth. Due to the multi-resolution hierarchy the amount of points that should be rendered is output-sensitive and the latency time is proportional to the maximum resolution of the mobile device. We have tested performance with wireless connection to an ADSL 8Mbps router, UMTS/HSPA and, in order to also include a worst-case situation, low-performance EDGE networks. In Table 2 we list the peak and average bandwidths, along with the time required to see a fully refined frame after application start-up. For these tests we typically set a screen tolerance of 2 pixels for adaptive refinement. With the iPad, at start-up we need to load 4.5MB of compressed data for full refinement of the David and 7.2MB for St. Matthew. Hence, the latency time is respectively about 1.5 and 2 seconds for a ADSL network connection, 3.5 and 6 seconds for UMTS/HSPA, and 225 and 375 seconds for EDGE. The iPhone has a resolution of 640 x 960, so at start-up it requires 250KB compressed data to fully refine the David model and 450KB for the St. Matthew. The latency times are less than 1 second for ADSL and UMTS/HSPA, while they are respectively about 12 and 22 seconds for EDGE.

8. Conclusions

We have presented a point-based rendering tool for local and remote interactive visualization of massive static point clouds on consumer-grade portable devices. The scalable data structure and framework architecture are able to cope with limitations in 3D graphics capabilities, memory, storage and network bandwidth, and they allow us to easily manage rendering of gigantic datasets on mobile platforms such as the Apple iPad and iPhone. We have tested our technique with a number of network technologies, showing how it is well-suited for on-site CH applications, such as exploration during scanning campaigns and the study, preservation or presentation of an artwork.

Acknowledgments. This work is partially supported by the EU FP7 Program under the DIVA project (REA Agreement 290277) and the INDIGO project (242341). We also acknowledge the contribution of Sardinian Regional Authorities. We thank Luca Pireddu for helpful comments and suggestions.

References

[BCK*11] BLUME A., CHUN W., KOGAN D., KOKKEVIS V., WEBER N., PETTERSON R., ZEIGER R.: Google body: 3d human anatomy in the browser. In *ACM SIGGRAPH 2011 Talks* (2011), ACM, p. 19. 2

[BGM*09] BETTIO F., GOBBETTI E., MARTON F., TINTI A., MERELLA E., COMBET R.: A point-based system for local and remote exploration of dense 3D scanned models. In *Proc. VAST* (2009), pp. 25–32. 2, 6

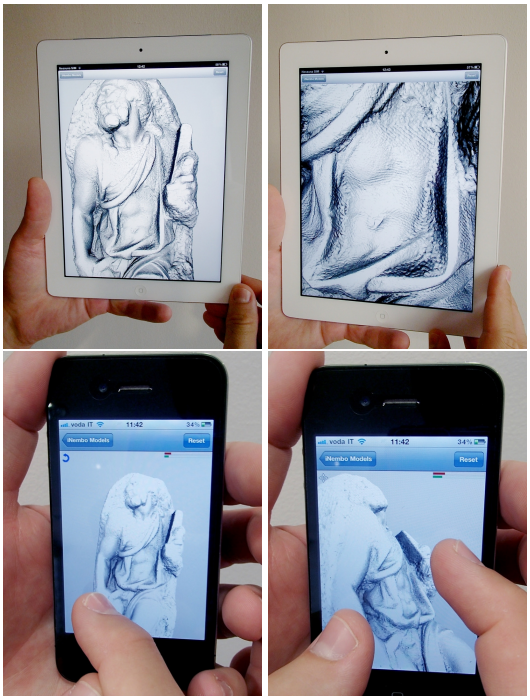


Figure 4: ***Colored St. Matthew dataset.*** *A 190M-point 3D model with ambient occlusion rendered with iPad and iPhone. The model is from the Digital Michelangelo project, courtesy of Marc Levoy and the Soprintendenza ai beni artistici e storici per le province di Firenze, Pistoia, e Prato.*

[BK03] BOTSCH M., KOBBELT L.: High-quality point-based rendering on modern gpus. In *Proc. Pacifig Graphics* (Oct. 2003), pp. 335–343. 6

[CPAM08] CAPIN T., PULLI K., AKENINE-MOLLER T.: The state of the art in mobile graphics research. *Computer Graphics and Applications, IEEE 28*, 4 (2008), 74–84. 2

[DD04] DUGUET F., DRETTAKIS G.: Flexible point-based rendering on mobile devices. *IEEE Computer Graphics and Applications 24*, 4 (July-August 2004). 2

[Eli75] ELIAS P.: Universal codeword sets and representations of the integers. *IEEE Trans. Inform. Theory 21*, 2 (Mar. 1975), 194–203. 3

[GEM*12] GOSWAMI P., EROL F., MUKHI R., PAJAROLA R., GOBBETTI E.: An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *The Visual Computer 28* (2012), 1–15. 2

[GKY08] GOBBETTI E., KASIK D., YOON S.-E.: Technical strategies for massive model visualization. In *Proc. ACM symposium on solid and physical modeling* (2008), pp. 405–415. 1, 2

[GM04a] GOBBETTI E., MARTON F.: Layered point clouds. In *Proc. Eurographics Symposium on Point Based Graphics* (2004), pp. 113–120,227. 2

[GM04b] GOBBETTI E., MARTON F.: Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics 28*, 1 (February 2004), 815–826. 2

[GMB*12] GOBBETTI E., MARTON F., BALSA RODRIGUEZ M., GANOVELLI F., DI BENEDETTO M.: Adaptive quad patches: an adaptive regular structure for web distribution and adaptive rendering of 3d models. In *Proc. ACM Web3D International Symposium* (August 2012), ACM Press, pp. 9–16. 2

[GP07] GROSS M., PFISTER H.: *Point-based graphics*. Morgan Kaufmann Pub, 2007. 1

[HL09] HE Z., LIANG X.: Multi-attributes controlled point-based rendering architecture for mobile devices. In *Computer-Aided Design and Computer Graphics, 2009. CAD/Graphics' 09. 11th IEEE International Conference on* (2009), IEEE, pp. 105–110. 2

[Hop97] HOPPE H.: View-dependent refinement of progressive meshes. In *Proc. SIGGRAPH* (1997), pp. 189–198. 2

[IST12] ISTI-CNR VISUAL COMPUTING LAB: MeshLab for iOS: A powerful easy-to-use 3D mesh viewer for iPad and iPhone. www.meshpad.org, 2012. 2

[JPP08] JOVANOVA B., PREDA M., PRETEUX F.: Mpeg-4 part 25: A generic model for 3d graphics compression. In *Proc. 3DTV* (2008), IEEE, pp. 101–104. 2

[KB04] KOBBELT L., BOTSCH M.: A survey of point-based techniques in computer graphics. *Computers & Graphics 28*, 6 (2004), 801–814. 2

[LE97] LUEBKE D., ERIKSON C.: View-dependent simplification of arbitrary polygonal environments. In *Proc. SIGGRAPH* (1997), pp. 199–208. 2

[LW85] LEVOY M., WHITTED T.: *The use of points as a display primitive*. Tech. Rep. TR 85-022, University of North Carolina at Chapel Hill, 1985. 2

[Mar12] MARION P.: Point cloud streaming to mobile devices with real-time visualization. www.pointclouds.org, 2012. 2

[MLL*10] MAGLO A., LEE H., LAVOUÉ G., MOUTON C., HUDELOT C., DUPONT F.: Remote scientific visualization of progressive 3d meshes with x3d. In *Proc. Web3D* (2010), pp. 109–116. 2

[MS03] MALAVAR H., SULLIVAN G.: YCoCg-R: A color space with RGB reversibility and low dynamic range. In *JVT ISO/IEC MPEG ITU-T VCEG*, no. JVT-I014r3. JVT, 2003. 3

[NKB10] NIEBLING F., KOPECKI A., BECKER M.: Collaborative steering and post-processing of simulations on hpc resources: Everyone, anytime, anywhere. In *Proceedings of the 15th International Conference on Web 3D Technology* (2010), ACM, pp. 101–108. 2

[PGC11] PINTUS R., GOBBETTI E., CALLIERI M.: Fast low-memory seamless photo blending on massive point clouds using a streaming framework. *ACM Journal on Computing and Cultural Heritage 4*, 2 (2011), Article 6. 6

[RL01] RUSINKIEWICZ S., LEVOY M.: Streaming QSplat: A viewer for networked visualization of large, dense models. In *Proc. Symposium on Interactive 3D Graphics* (2001). 2

[SLDJ04] SENECAL J. G., LINDSTROM P., DUCHAINEAU M. A., JOY K. I.: An improved N-bit to N-bit reversible Haar-like transform. In *12th Pacific Conference on Computer Graphics and Applications* (Oct. 2004), pp. 371–380. 3

[WBB*08] WAND M., BERNER A., BOKELOH M., JENKE P., FLECK A., HOFFMANN M., MAIER B., STANEKER D., SCHILLING A., SEIDEL H.-P.: Processing and interactive editing of huge point clouds from 3D scanners. *Computers & Graphics 32*, 2 (April 2008), 204–220. 2, 6

[WK02] WU J., KOBBELT L.: Fast mesh decimation by multiple-choice techniques. In *Procs of 7th International Fall Workshop on Vision, Modeling, and Visualization* (2002), pp. 241–248. 3

[WS06] WIMMER M., SCHEIBLAUER C.: Instant points. In *Proc. Symposium on Point-Based Graphics* (July 2006), pp. 129–136. 6

[XV96] XIA J., VARSHNEY A.: Dynamic view-dependent simplification for polygonal models. In *Proc. IEEE Visualization* (1996), pp. 327–334. 2