On-the-fly Appearance Quantization on GPU for 3D Broadcasting

Julien Hadim, Tamy Boubekeur, Mickaël Raynaud, Xavier Granier, Christophe Schlick

On-The-Fly Appearance Quantization on the GPU for 3D Broadcasting

Julien Hadim Tamy Boubekeur Mickaël Raynaud Xavier Granier Christophe Schlick
IPARLA project (INRIA Futurs - LaBRI*)

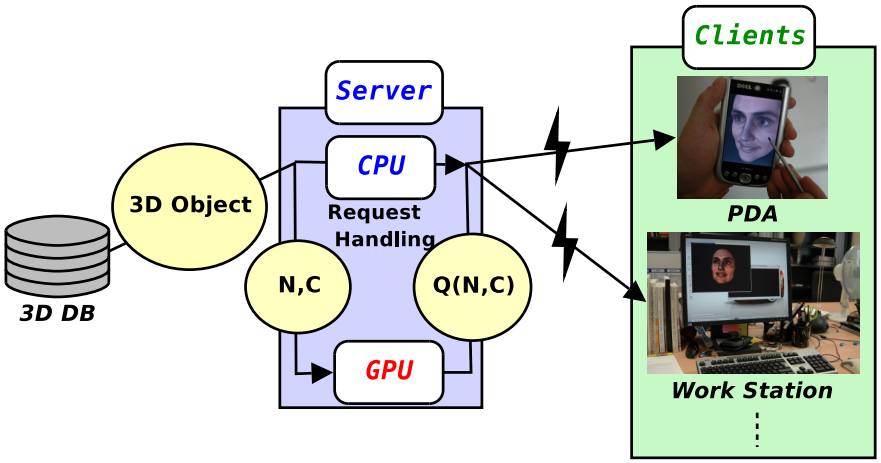


Figure 1: *A client/server 3D broadcasting application, with on-demand quantization to reduce bandwidth. The on-demand quantization workload is delayed to the GPU on the server side.*

Abstract

This paper presents an improved client-server system that increases the availability of remote 3D data. In order to reduce the required bandwidth, the data related to the appearance (color and normal) involved in the rendering of meshes and point clouds is quantized on-the-fly during the transmission to the final client, without reducing the geometric complexity. Our new quantization technique for the appearance that can be implemented on the GPU, strongly reduces the CPU load on the server-side and the transmission time is largely decreased.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/server;

Keywords: Quantization, appearance, huge models, streaming, GPU

1 Introduction

In order to provide an efficient distribution of 3D content with a client/server approach, the availability of the remote data has to be taken into account. One classic solution to lower the effects of network bottlenecks or latencies, is to limit the required bandwidth by reducing the complexity of the 3D models. Generally this simplification is combined with an adaptation to the different 3D capacities of the clients. Thus, one of the two following solutions is usually chosen:

- Several levels of detail of the model are stored and the selection of the adapted resolution to be sent is done according to the capacity of the client. This solution is simple and does not require on-the-fly processing but implies to create, store, and manage multiple versions of each 3D object.
- All the objects of the database are converted into a multiresolution representation. For each request, one or more levels of details to be sent are filtered out according to the clients' capacity. This solution reduces the storage overhead compared to the previous approach but requires a pre-processing step which might be relatively complex.

A solution that combines the advantages of the previous ones would be to store only the full resolution of each object and to build the level of details on demand. Unfortunately, the resulting processing slows down when more than a few clients request the 3D database at the same time.

All these solutions focus only on reducing the data complexity for an adaptation to a client. Nowadays, thanks to the evolution of graphics hardware, even mobile devices are now able to display thousands to millions of polygons per second (cf. Figure 2). On the other side, reduced network bandwidth and latency remain critical issues in many situations. Thus, the problem of 3D broadcasting is less related to the geometric complexity that the client is able to manage rather than the size of data to transmit.

In this context, it is possible to use a special kind of compression for the transmitted data that is very simple and efficient: *data quantization*. Compared to the other compression techniques, quantization has two strong advantages:

- Its compression data gain is determined in advance.
- Its local behavior makes it easily parallelizable.

*UMR CNRS 5800 with University of Bordeaux and ENSEIRB.

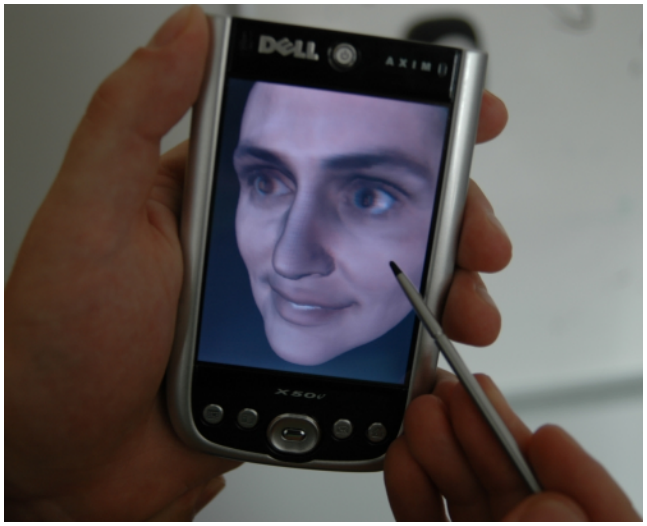


Figure 2: *The power of recent mobile terminals makes them appealing as potential clients for 3D broadcasting. On this figure, interactive dequantization and rendering are achieved on a PDA with OpenGL ES capacity.*

In recent years, the ubiquitous use of 3D acquisition devices (e.g. laser range scanners) in computer graphics has led to a strong evolution of 3D data representation. The complexity of many acquired models makes it difficult to generate a consistent parameterization that could be used to apply standard texture mapping techniques. Usually, for point sampled models, each vertex is combined with a local description of its appearance classically expressed as a diffuse color and a normal vector. Our goal is to provide on-the-fly quantization of a 3D stream (defined as per vertex position, normal and color), in order to reduce transmission time. For such models, the appearance represents two third of the total object size to finally produce the 24 bits color of a pixel for standard displays (16 bits on most of the PDAs and cell-phones). Hence, quantizing these two attributes would significantly reduce the data size. Unfortunately, even the most simple quantization techniques require a non-negligible amount of processing time, which introduces latency in the data access.

In order to reduce this latency, we propose an on-the-fly quantization of the appearance (normal and color) that is designed to be processed on the GPU (cf. Figure 1). The normal and color attributes are quantized by the GPU on the server-side. This reduces the CPU workload for input/output accesses. Delaying the quantization on the GPU exhibits two major advantages:

- A minimum gain of 20% for the quantization processing time
- A CPU workload reduction so as a larger number of simultaneous requests that can be handled.

The combined reduction of the CPU workload and the overall data size significantly increases the availability of the 3D data. Thanks to the on-the-fly process, this quantization can be easily introduced during the transmission to the final client and is thus compatible with other complexity reduction techniques. With such an approach, the storage of all the quantized levels of details is not required, and the complexity of the original models is preserved.

The remainder of this paper is organized as follow: Section 2 introduces the previous work related to our context; Section 3 presents the general overview of our system; Section 4 details the principles of our GPU quantization; Section 5 details implementation issues, and Section 6 provides experimental results in terms of performance and quality. Finally, Section 7 concludes and proposes future improvements of our framework.

2 Previous Work

When it comes to size reduction of 3D data objects, one classical solution is to perform mesh decimation (e.g., [Lindstrom 2000; Lindstrom and Silva 2001; Wu and Kobbelt 2003]). These approaches simplify large meshes by applying vertex clustering or edge collapsing. Unfortunately, this process also reduces the geometric quality of the final mesh. Another approach, introduced by Hoppe [1996], proposes a different representation consisting in a coarse mesh with several refinement steps to retrieve the original resolution. This method is well adapted for streaming data [Prince 2000; Pajarola and Rossignac 2000]. All these approaches are restricted to meshes and require costly preprocessing. Moreover, they only address the problem of size reduction for the geometry, but not for the appearance.

Some global optimization techniques have been developed (e.g., [Purnomo et al. 2005]) in order to provide the optimal quantization code for all the vertex attributes. Intrinsically global, such a method can hardly process streamed data. The most related approach is the solution proposed by Rusinkiewicz et Levoy [2001]. This solution offers a multiresolution approach based on a preprocessing and a lower data size. But this solution is only valid for point-based rendering.

A natural approach is the quantization of each normal coordinate on a limited number of bits (e.g., [Bajaj et al. 1999]). Easy to implement, this solution is neither uniform nor isotropic. Moreover, it does not preserve the unit length of the corresponding vectors. The number of bits can be optimized by clustering [Kim et al. 2004], but an object-dependent look-up table is then needed for dequantization step.

Normal quantization

The octahedron-based quantization, proposed by Deering [Deering 1995], has been widely used to obtain a more uniform normal compression. The direction space is divided in 8 regions according to an octahedron. Each region is further subdivided in 6 sections, and for each section, the angles are encoded using a regular grid. Deering stated that $100,000$ directions (a maximum angular density of 0.01) are sufficient for normal quantization without visible artifacts, if the directions are uniformly and isotropically distributed. Since the regular grid does not provide such a distribution, he had to switch to $200,000$ normals, using an 18-bit encoding, in order to get artefact-free quantization.

In the work of Taubin et al. [1998], a regular quad-tree is used on each face of the octahedron. The encoding and decoding are thus recursive, and the resulting normal has to be normalized. The main advantage of this approach is a more uniform distribution and an efficient usage of all the bits used for the encoding. In the method proposed by Botsch et al. [2002], the authors claim that a 13-bit encoding (i.e. $8,192$ directions) provides sufficient quality for point-based rendering.

In QSplat [Rusinkiewicz and Levoy 2000], a cube is used as the support of quantization. The quantized normals are obtained by applying a 52×52 grid on each of the 6 faces, combined with a warping function to sample the direction space more uniformly. A global look-up table of $16,224$ entries is used for the decompression, resulting in a 14-bit encoding.

3 Overview of the system

3.1 Server Side

The general architecture of our system (cf. Figure 1) is build around a 3D broadcasting server. As usual, this server is connected to a database of 3D objects and distributes them. For coherency reasons, our database provides one single kind of 3D object: dense point clouds coming, for instance, from 3D acquisition devices. These point clouds are formed with chunks of 9 floating-point values (3 for the position, 3 for the normal, and 3 for the color). By default, incoming data is considered as unorganized (e.g. raw data from the acquisition pipeline). However, if a reconstructed mesh is provided, the geometry is encoded similarly, and the connectivity is simply encoded through a special flag which indicates that the point cloud order corresponds to a single *degenerated strip*. This encoding *implicitly* defines a polygonal mesh over the point cloud. This approach lets us process only point clouds, while maintaining compatibility with meshes. It just requires binding a different rendering mode on client side. It is thus compatible with any process for streaming 3D content that transfers vertex position with the corresponding normal and color.

The originality of our system is to the use of a GPU on the server-side. The stream coming from the data-base during a query is divided in *positions* and *appearance properties* (normals and colors). The positions are currently transmitted without processing, while the appearance properties are transferred to the GPU, where these values are quantized in a GPU-friendly process (see Section 4). Then, the client receives data with original positions and quantized appearance properties.

The GPU-stream is buffered, decomposing the original object in subsets of a few hundred thousand samples batch-processed by fragment shading. The whole system can run out-of-core by processing iteratively only a small sub-part of the object, consequently the memory foot-print required by a given query is strongly reduced which increases the number of simultaneous queries supported by the server. Once quantized, this part of the model is transmitted to the client.

Note that, on the server side, a caching scheme can be used to save the quantized data in order to accelerate the subsequent transmission of highly requested 3D object, similar to the caching scheme commonly used for web pages with dynamic contents.

3.2 Client Side

When a client sends a request for a 3D model, the geometric size is first transmitted in order to initialize the object structure. The incoming streams are then dequantized when received, in order to fill up this structure. By this progressive transfer, the object can be partially rendered or rendered only when the transmission is completed.

4 Fast Appearance Quantization

To quickly quantize the appearance of each geometry sample, we propose to quantize normal vectors and vertex colors by using efficient methods which can be easily adapted to work on the GPU. We use a similar two step approach to quantize either the normal or the color:

- All possible quantized values are precomputed and stored in an adapted array structure, respectively a 2D grid for normals and a 3D grid for colors.

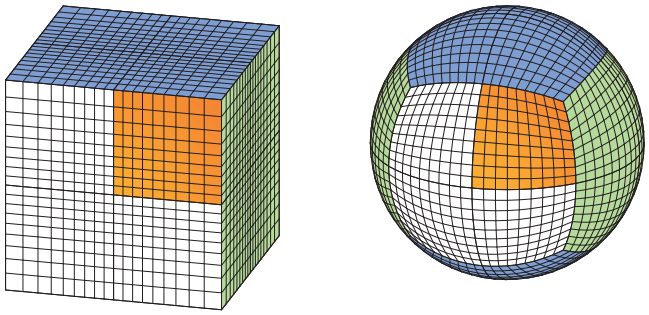


Figure 3: *Original warping lines on the unit cube (left) and their projection to the unit sphere (right), the orange zone corresponds to the only required part of the sphere to be represented in the look-up table due to symmetry reasons*

- The values of the normal/color to be quantized are then used as indexes to retrieve the corresponding quantized value into the array structure.

This method allows a very fast quantization since all quantized values are precomputed and the runtime quantization process is a simple access to the array structure. By implementing both array structures as textures, the whole process can be straightforwardly implemented on the GPU. Moreover, using a higher resolution for the array structure than the quantized space allows one to generate non-linear sampling during the access to the array structure.

Finally, we combine normal quantization with color quantization in order to compress the appearance of models in a 32-bit word. More precisely, we quantize normals with 17 bits and color with 15 bits, to stay compatible with current hardware internal formats used for 16 bits precision (`GL_UNSIGNED_SHORT_5_6_5`).

4.1 Normal Quantization

Since there is no global uniform and isotropic parameterization over the surface of a unit sphere, representing the direction space, the most efficient method to quantize the directions is to pre-sample this space as uniformly and isotropically as possible and associate to each sample the floating point representation with a look-up table (**LUT** for short). In order to reduce the huge size of this LUT, the spherical domain can be partitioned by taking into account the natural symmetries it contains. One can easily show [Deering 1995] that only $1/24^{th}$ of the samples require actual storage. The remainder can be retrieve by symmetry or index permutation. Our normal quantization uses a cube-based approach, where each face is warped to a more uniform distribution of normals, as in QSplat [Rusinkiewicz and Levoy 2000]. We use face naming and indexing of the hardware cube-map definition $\pm X, \pm Y, \pm Z$. By mapping an axis-aligned unit cube into the unit sphere, we obtain six identical spherical sections (see Figure 3-left). Moreover, each spherical section includes two axial symmetries that correspond to the horizontal and vertical symmetries in the local space of the section. Thus, the LUT can be restricted to one quarter of the cube face which corresponds to $1/24^{th}$ of the whole sphere (orange area in Figure 3-left). Each pixel of the 2D grid contains the coordinates of floating-point unit length normals. Each portion of the sphere associated with a cube face is almost uniformly sampled by a parameterized grid using a warping function $f(t) = \tan(4t/\pi)$ on the parameterization (u,v) of the corresponding cube face:

$$\begin{array}{ll} \forall u \in [-1,1] & u' = f(u) \in [-1,1] \\ \forall v \in [-1,1] & v' = f(v) \in [-1,1] \end{array}$$

Algorithm 1 Dequantization

```
func decode(quantizedNormal)
    Decode index, face and symmetries HS and VS
    Retrieve x,y,z from xyzLookUp[index]
    if (VS) then z = -z
    if (HS) then y = -y
    switch (face)
        case '±X' : return (±x, y, z)
        case '±Y' : return (z, ±x, y)
        case '±Z' : return (z, y, ±x)
```

For each sample of the quantized direction space, we need a bit code to retrieve floating representation from the reduced LUT. The face and the symmetries (i.e., the 24 different positions of the LUT on the sphere) require 5 bits to be encoded. For simplicity, the combinations are encoded as bit-masks:

- The cube face is encoded on three bits.
- The two axial symmetries (VS and HS) on one bit each.

Once the 5-bits "header" has been determined, the last bits are used to define a linear index in the LUT. More precisely, for a k^2 resolution, each texel (i, j) corresponds to the index $i \times k + j$. We propose to use 6 bits to encode i and j respectively. Thus by using a LUT of 4096 entries, we can store as many as $98,304$ different normals. Note that this is very close to the $100,000$ normals recommended by Deering [Deering 1995].

4.2 Normal Dequantization

The dequantization process consists in one access to the normal LUT and at the most three inversions and two permutations between the coordinates of the normal (see Algorithm 1). First, the index is decoded in order to access to the value of the normal in the LUT. Secondly, the face and the two symmetry flags are extracted to get the final positioning of the normal in the correct spherical section. To attenuate quantization artifacts that may appear, for example, as band effects in large areas of low curvature, we can use jittering to randomly perturb the normal coordinates in order to convert band effects into less objectionable high frequency white noise [Cook et al. 1984]. Since our representation is nearly uniform, the range of the added noise can be easily computed from the quantization step.

4.3 Color Quantization/Dequantization

Our color quantization approach uses a 3D color grid where each voxel encodes a quantized color. The float color values are used as coordinates in this color volume to access the corresponding quantized color. Each voxel can encode a color on 15 bits with respectively 5 bits for each component. We have restricted our color quantization to 15 bits in order to easily combine it with our normal quantization on 17 bits. So the whole quantized appearance information can be stored on two 16-bit words, as shown in Figure 4. As in the case of normal quantization, if we increase the resolution of the color grid, several voxels may correspond to a unique color.

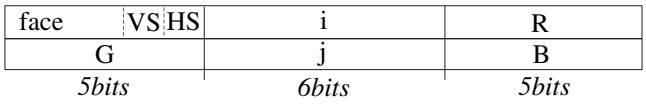


Figure 4: *Appearance parameters are stored on* 32 *bits. Normals are encoded on* 17 *bits (5 bits for face and symmetries, 6 bits for i, 6 bits for j), and colors are encoded on* 15 *bits (5 bits for red, green and blue, respectively).*

This property allows the construction of the color grid with an alternative color space sampling instead of simple linear quantization.

5 GPU Implementation

We have tested our appearance compression on GPU for geometric models providing per-vertex normals and colors. But, as detailed above, the encoding of the appearance attributes is based on the `5_6_5` texture format. The conformity to this format may also permit, in the case of a parameterized mesh, to specify appearance attributes into textures rather than in a per-vertex scheme. The dequantization algorithm needs some specific modifications in order to be adapted to the vertex unit and the fragment unit of the GPU.

5.1 Encoding

The main contribution of our method is an out-of-core encoding of streamed data (without having to store the full object in memory at any moment). The processing loop is subdivided into three steps:

1. Read a fixed-size data set from the input stream (file or socket) and store it into a vertex buffer
2. Process the quantization step in the fragment units on the GPU
3. Write results into the output stream (file or socket).

The quantization process results in a simple cube-map texture access on the GPU and this operation only depends on fragment units, so the data to encode is transmitted with textures on the GPU. The quantization results are read-back from the current frame-buffer.

The encoding is reduced to the rendering of a textured quad with the same size for the input texture and the framebuffer. By this way, we ensure a bijection between each input texel (normal or color that has to be quantized and that has been stored in a texture) and each output pixel (corresponding quantized data). For each new normal/color buffer, we render an image and the result is directly written to the output stream. The system thus runs with constant memory impact, which allows us the quantization of several objects in parallel on a same server: the bottleneck is thus only limited by the input/output, the CPU being totally freed from the quantization process.

5.2 Input/output transfer optimization

In order to reduce the memory transfers between CPU and GPU, we have developed a single pass solution. For a buffer size of $B \times B$, the color and normal to be quantized are transferred to the GPU using a $2B \times B$ texture (cf. Figure 5. Color and normal values are thus interlaced in this input texture, and the selection of the quantization process is done on the fragment processor (using alternatively color quantization and normal quantization). At the end of the process, the GPU generates a similar $2B \times B$ framebuffer that contains interlaced quantized normals and colors. Note that, with this buffered and purely local approach, multiple objects can be quantized in parallel. One simple solution is to combine data from different objects into a single buffer. Another approach, as batch processing scheme, would be to buffer alternatively each object at each frame rendered. With theses solutions, multiple requests could be easily supported.

5.3 Dequantization

The normal and color dequantization takes place on the client side. In the case of CPU dequantization, we directly use the algorithm described previously (cf. Algorithm 1) for normal dequantization. The color dequantization is even easier as it simply consist to switch

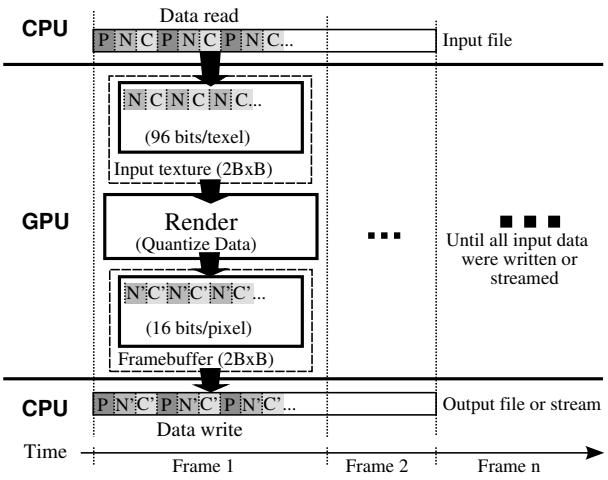


Figure 5: *For each rendered frame on the GPU, floating-point normals and colors (resp. N and C, P being the position) are read from input file or stream and send interlaced to the GPU through a floating-point texture. After rendering, quantized normals and colors (res. N' and C') are read-back from the framebuffer and written into output file or stream.*

from a 15 bits to a 24 bits RGB encoding, by performing a 3 bit left shift on each color component.

By using some specific adaptations, this algorithm can even be implemented in the vertex unit of the GPU. Since the LUT is too large to fit in the vertex registers as a static table, and since vertex texturing is slow on current GPUs, we use an 8×8 LUT combined with a linear blending. For a given index at full LUT size (64×64 for quantized the normal on 17 bits), the four nearest encoded normals are determined and decoded with the LUT. The normal is obtained with linear blending of the four nearest values, and normalized to reduce quantization artifacts. To decode face and symmetry flags, we use a vector table containing the 24 configurations encoded in the 5 bits header. Each value of this table is a vector storing separately the face and the two symmetries.

Unfortunately, current GPUs do not support bit-wise operators or native modulus. These operators can be emulated using the technique described in [Purnomo et al. 2005] which shows how to use fractional and floor functions to simulate bit masks (more details in [Purnomo et al. 2005]). In practice, this solution is too slow to be competitive with a CPU solution. With the brand new of bit-wise operators [NVIDIA 2006], efficient GPU implementations of our scheme will soon be possible.

As said above, to reduce quantization artifacts, jittering can be added both on the normal and on the color. One easy way to obtain color jittering, is to jitter the blending coefficient used during Gouraud shading. When using Phong shading, this process also jitters the normals. Implementing jittering for CPU dequantization is quite easy, but rather tricky on GPU because it is still complex to simulate a simple random generator on existing GPUs.

In practice, our system only uses CPU dequantization since our GPU implementation of dequantization algorithm is not competitive for interactive rendering due to current hardware limitations. On the client side, the main benefit of quantization is to reduce the memory footprint by storing 3D data in quantized form.

Size (kB)	64	128	256	512	640
Time 1 (ms)	747	781	770	764	764
Time 2 (ms)	731	622	564	453	419
Time 3 (ms)	371	467	592	610	579

Table 1: *Impact of the buffer size on the quantization speed. The timings on the first row are obtained with a pure CPU implementation on a workstation with a 3.4 GHz Pentium 4. The timings on the second row are obtained with a GPU implementation on the same workstation with a Quadro FX3400 on a PCI Express x16 bus. Finally, the timings on the third row are obtained with a GPU implementation on a laptop with a 2.26 GHz Pentium-M and a GeForce 7800GTX on a PCI Express x16 bus. .*

6 Results and Discussion

To evaluate the benefit of our system, we first compare the quantization time when using both the CPU and the GPU implementation of our scheme. Then we compare the required client/server transfer time between the original data and the quantized data including required time to quantize on server side and dequantize on client side. Finally, we provide a visual comparison between a model with its original appearance precision and our corresponding version. The workstation acting as a server is a 3GHz Pentium 4 with 1GB RAM, a NVIDIA Quadro 4400 graphics running Windows XP.

6.1 Quantization benchmarks

As explained above, our system runs out-of-core by using a buffered reading of the input stream. According to the implementation, each buffer is then either directly processed on the CPU or transferred to the GPU. Table 1 illustrates the impact of the buffer size on the quantization speed. Actually, it appears that this buffer size has only a marginal impact when performing a pure CPU quantization (whatever the buffer size, the quantization takes about 765 ms on our reference workstation). On the other hand, we notice that, in the case of GPU implementation, the optimal buffer size depends greatly on the configuration and has to be carefully chosen according to the server capacities. This is not really surprising, because it is well known that when using GPU for general purpose computation, memory transfer to and from the CPU usually represents the major bottleneck.

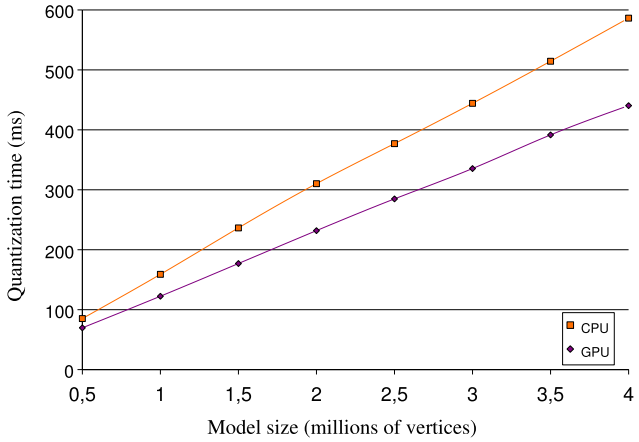


Figure 6: *GPU vs. CPU quantization times for appearance on point cloud models*

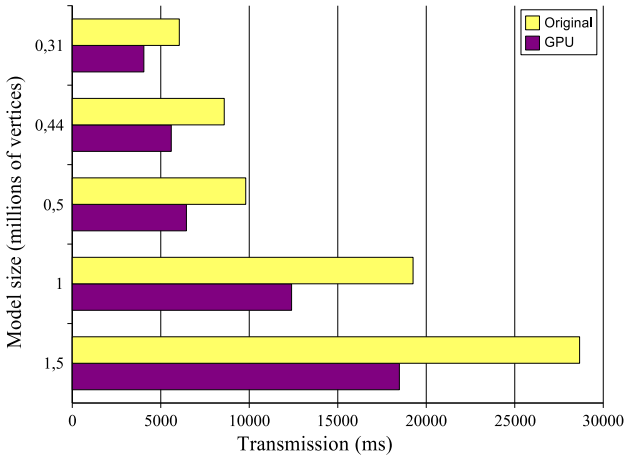


Figure 7: *Average time required to transmit a model (original and quantized on the GPU). For the quantized model, the results include the quantization and the dequantization. The measures have been performed on a WIFI connection at 54Mb/s.*

We then analyze the quantization speed for models of increasing size, with and without using the GPU implementation (cf. Figure 6). As can be seen, the use of the GPU improves the quantization speed from 20 % to 25% in general. So, the use of the GPU not only reduces the CPU workload, but significantly speeds up the quantization process.

6.2 Streaming benchmark

We have tested our client-server configuration in a mobility context, using our laboratory WIFI connection at 54Mb/s. As it was not a wireless connection dedicated to our experiment, we have computed the average value of the transmission time over several measures, in order to reduce the bias due to possible network congestion. For these tests, the client was a laptop with a Pentium-M 2.26 GHz. Thanks to a fast quantization process, the transmission is largely reduced (about 35% faster - cf. Figure 7). In the case of quantized data, the transmission time includes the quantization on the server and the dequantization of the client.

We have also tested the effect on transmission time when using either the GPU quantization or the CPU one. Since the quantization is significantly faster on the GPU (cf. Figure 6), we expected a corresponding benefit on the whole transmission time. Surprisingly, this is not the case. We have noticed that the transmission is actually slightly slower when using the GPU implementation (2% in average, and decreasing with the increasing size of the object). After further investigations, we think that, even if the CPU workload is reduced, more data transfer is performed on the server (transfers from disk, between GPU and CPU and to the network cards instead of only transfers from disk to the network cards through the CPU). With the upcoming multi-core processors, combining GPU and CPU, the overhead involved by these internal data transfer is likely to be canceled.

6.3 Quantization Error

Figure 8 shows an average size model (8M polygons) rendered with a specular material, with several light sources. The RGB error given is multiplied by 100. We only observe a small loss between quantized perception and original one. By measuring perceptual error *Delta E* (CIE 1976) on the same image, we obtain an average error of 8.8 10^{-2}% with a maximum error of 9.7% (error divided by maximum possible error when colored components vary between 0 and 1 in RGB space).

7 Conclusion and Future Work

We have proposed a scheme for quantization of appearance properties designed for client-server 3D broadcasting applications. In order to reduce the CPU workload as much as possible, the quantization process runs completely on the GPU, via a streaming process. Our out-of-core method is totally insensitive to the size of the 3D model (i.e., a bounded memory footprint can be guaranteed). We have also introduced a fast combination of octahedron-based and cube-based quantization methods. Our solution significantly improves quantization capacities of the server and thus allows a higher number of simultaneous requests for the visualization of huge 3D online databases on heterogeneous clients.

In future work, we plan to move a maximum of the geometric processing on the server side to the GPU. More precisely, we are working on the simplification and quantization of the geometry, leading to an on-the-fly generation of level of details. In order to increase the efficiency of such an approach, we are also working on better management of memory transfers on the server side. Furthermore, in order to limit the dequantization cost required on the client side, we are working on illumination models in the quantized normal space.

Acknowledgment

We would like to thank the Region Aquitaine for the funding support of this project, and all the reviewers for their helpful comments.

References

BAJAJ, C. L., PASCUCCI, V., AND ZHUANG, G. 1999. Single Resolution Compression of Arbitrary Triangular Meshes with Properties. In *Proc. IEEE Conference on Data Compression '99*, 247.

BOTSCH, M., WIRATANAYA, A., AND KOBBELT, L. 2002. Efficient High Quality Rendering of Point Sampled Geometry. In *Proc. EUROGRAPHICS workshop on Rendering 2002*, 53–64.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 137–145.

DEERING, M. 1995. Geometry Compression. In *Proc. ACM SIGGRAPH '95*, 13–20.

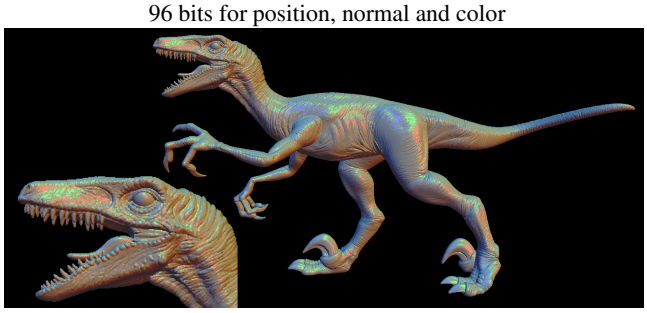
HOPPE, H. 1996. Progressive meshes. *Computer Graphics 30*, Annual Conference Series, 99–108.

KIM, D.-S., CHO, Y., AND KIM, H. 2004. Normal vector compression of 3d mesh model based on clustering and relative indexing. *Future Gener. Comput. Syst. 20*, 8, 1241–1250.

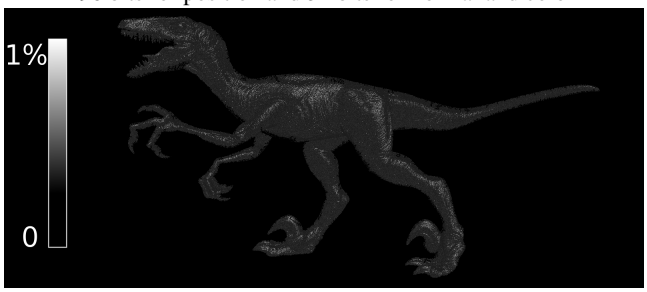
LINDSTROM, P., AND SILVA, C. 2001. A memory insensitive technique for large model simplification. In *Proceedings of IEEE Visualization 2001*, 121–126.

LINDSTROM, P. 2000. Out-of-core simplification of large polygonal models. In *Siggraph 20000, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 259–262.

Original model - 288 bits/vertex
96 bits for position, normal and color



Quantized normals and colors on GPU - 128 bits/vertex
96 bits for position and 32 bits for normal and color



Color error - Difference image x100

Figure 8: *Quantization error in Phong shading. Three light sources, specular surface.*

NVIDIA, 2006. Opengl extensions for geforce 80, Nov. http://developer.nvidia.com/object/nvidia_opengl_specs.html.

PAJAROLA, R., AND ROSSIGNAC, J. 2000. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics 6*, 1 (/), 79–93.

PRINCE, C. 2000. *Progressive Meshes for Large Models of Arbitrary Topology*. PhD thesis, University of Washington.

PURNOMO, B., BILODEAU, J., COHEN, J. D., AND KUMAR, S. 2005. Hardware-Compatible Vertex Compression Using Quantization and Simplication. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 53–61.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: a multiresolution point rendering system for large meshes. In *Proc. ACM SIGGRAPH '00*, 343–352.

RUSINKIEWICZ, S., AND LEVOY, M. 2001. Streaming QSplat: a viewer for networked visualization of large, dense models. In *Proc. Symposium on Interactive 3D graphics 2001*, 63–68.

TAUBIN, G., HORN, W., LAZARUS, F., AND ROSSIGNAC, J. 1998. Geometry coding and VRML. *Proceedings of the IEEE 86*, 6, 1228–1243.

WU, J., AND KOBBELT, L. 2003. A stream algorithm for the decimation of massive meshes. In *Graphics Interface'03 Conference Proceedings*, 185–192.